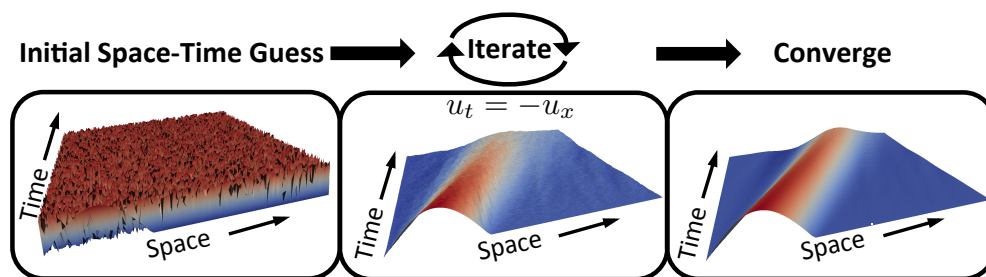


XBraid

Parallel Multigrid in Time



V. A. Dobrev, R. D. Falgout, Tz. V. Kolev, N. A. Petersson, J. B. Schroder, U. M. Yang
Center for Applied Scientific Computing (CASC)
Lawrence Livermore National Laboratory

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-SM-660398

Copyright (c) 2013, Lawrence Livermore National Security, LLC. Produced at the Lawrence Livermore National Laboratory. Written by the XBraid team. LLNL-CODE-660355. All rights reserved.

This file is part of XBraid. Please see the COPYRIGHT and LICENSE file for the copyright notice, disclaimer, and the GNU Lesser General Public License. Email xbraid-support@llnl.gov for support.

XBraid is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License (as published by the Free Software Foundation) version 2.1 dated February 1999.

XBraid is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the IMPLIED WARRANTY OF MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the terms and conditions of the GNU General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Contents

1 Abstract	1
2 Introduction	2
2.1 Meaning of the name	2
2.2 Overview of the XBraid Algorithm	2
2.2.1 Two-Grid Algorithm	6
2.2.2 Summary	6
2.3 Overview of the XBraid Code	7
2.3.1 Parallel decomposition and memory	7
2.3.2 Cycling and relaxation strategies	8
2.3.3 Overlapping communication and computation	9
2.3.4 Configuring the XBraid Hierarchy	9
2.3.5 Halting tolerance	10
2.3.6 Heat equation example	11
2.4 Citing XBraid	12
2.5 Summary	12
3 Example	12
4 Building XBraid	18
5 Compiling and running the examples	18
6 Coding Style	19
7 Using Doxygen	19
8 Regression Testing	20
9 Module Index	24
9.1 Modules	24
10 Data Structure Index	25
10.1 Data Structures	25
11 File Index	25
11.1 File List	25
12 Module Documentation	25
12.1 User-written routines	25

12.1.1 Detailed Description	26
12.1.2 Typedef Documentation	26
12.2 User interface routines	29
12.2.1 Detailed Description	29
12.3 General Interface routines	30
12.3.1 Detailed Description	30
12.3.2 Typedef Documentation	30
12.3.3 Function Documentation	31
12.4 XBraid status routines	38
12.4.1 Detailed Description	39
12.4.2 Macro Definition Documentation	39
12.4.3 Typedef Documentation	39
12.4.4 Function Documentation	39
12.5 XBraid test routines	46
12.5.1 Detailed Description	46
12.5.2 Function Documentation	46
13 Data Structure Documentation	51
13.1 _braid_AccessStatus Struct Reference	51
13.1.1 Detailed Description	51
13.1.2 Field Documentation	51
13.2 _braid_AccuracyHandle Struct Reference	51
13.2.1 Detailed Description	52
13.2.2 Field Documentation	52
13.3 _braid_CoarsenRefStatus Struct Reference	52
13.3.1 Detailed Description	53
13.3.2 Field Documentation	53
13.4 _braid_CommHandle Struct Reference	53
13.4.1 Detailed Description	53
13.4.2 Field Documentation	53
13.5 _braid_Core Struct Reference	54
13.5.1 Detailed Description	55
13.5.2 Field Documentation	55
13.6 _braid_Grid Struct Reference	58
13.6.1 Detailed Description	59
13.6.2 Field Documentation	59
13.7 _braid_PhiStatus Struct Reference	61

13.7.1 Detailed Description	61
13.7.2 Field Documentation	61
14 File Documentation	61
14.1 _braid.h File Reference	61
14.1.1 Detailed Description	62
14.1.2 Macro Definition Documentation	63
14.1.3 Function Documentation	63
14.1.4 Variable Documentation	66
14.2 braid.h File Reference	66
14.2.1 Detailed Description	67
14.3 braid_defs.h File Reference	68
14.3.1 Detailed Description	68
14.3.2 Macro Definition Documentation	68
14.3.3 Typedef Documentation	68
14.3.4 Variable Documentation	69
14.4 braid_status.h File Reference	69
14.4.1 Detailed Description	70
14.5 braid_test.h File Reference	70
14.5.1 Detailed Description	70
14.6 util.h File Reference	70
14.6.1 Detailed Description	71
14.6.2 Function Documentation	71
Index	72

1 Abstract

This package implements an optimal-scaling multigrid solver for the (non)linear systems that arise from the discretization of problems with evolutionary behavior. Typically, solution algorithms for evolution equations are based on a time-marching approach, solving sequentially for one time step after the other. Parallelism in these traditional time-integration techniques is limited to spatial parallelism. However, current trends in computer architectures are leading towards systems with more, but not faster, processors, i.e., clock speeds are stagnate. Therefore, faster overall runtimes must come from greater parallelism. One approach to achieve parallelism in time is with multigrid, but extending classical multigrid methods for elliptic operators to this setting is a significant achievement. In this software, we implement a non-intrusive, optimal-scaling time-parallel method based on multigrid reduction techniques. The examples in the package demonstrate optimality of our multigrid-reduction-in-time algorithm (MGRIT) for solving a variety of equations in two and three spatial dimensions. These examples can also be used to show that MGRIT can achieve significant speedup in comparison to sequential time marching on modern architectures.

It is **strongly recommended** that you also read [Parallel Time Integration with Multigrid](#) after reading the [Overview of the XBraid Algorithm](#). It is a more in depth discussion of the algorithm and associated ex-

periments.

2 Introduction

2.1 Meaning of the name

We chose the package name XBraid to stand for *Time-Braid*, where X is the first letter in the Greek work for time, *Chronos*. The algorithm *braids* together time-grids of different granularity in order to create a multigrid method and achieve parallelism in the time dimension.

2.2 Overview of the XBraid Algorithm

The goal of XBraid is to solve a problem faster than a traditional time marching algorithm. Instead of sequential time marching, XBraid solves the problem iteratively by simultaneously updating a space-time solution guess over all time values. The initial solution guess can be anything, even a random function over space-time. The iterative updates to the solution guess are done by constructing a hierarchy of temporal grids, where the finest grid contains all of the time values for the simulation. Each subsequent grid is a coarser grid with fewer time values. The coarsest grid has a trivial number of time steps and can be quickly solved exactly. The effect is that solutions to the time marching problem on the coarser (i.e., cheaper) grids can be used to correct the original finest grid solution. Analogous to spatial multigrid, the coarse grid correction only *corrects* and *accelerates* convergence to the finest grid solution. The coarse grid does not need to represent an accurate time discretization in its own right. Thus, a problem with many time steps (thousands, tens of thousands or more) can be solved with 10 or 15 XBraid iterations, and the overall time to solution can be greatly sped up. However, this is achieved at the cost of more computational resources.

To understand how XBraid differs from traditional time marching, consider the simple linear advection equation, $u_t = -cu_x$. The next figure depicts how one would typically evolve a solution here with sequential time stepping. The initial condition is a wave, and this wave propagates sequentially across space as time increases.

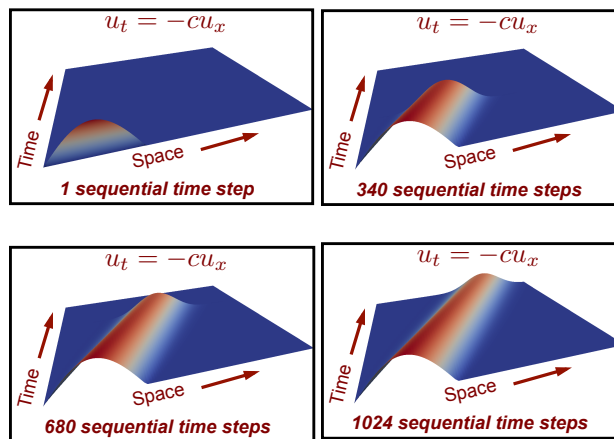


Figure 1: Sequential time stepping.

XBraid instead begins with a solution guess over all of space-time, which for demonstration, we let be random. An XBraid iteration does

1. Relaxation on the fine grid, i.e., the grid that contains all of the desired time values Relaxation is just a local application of the time stepping scheme, e.g., backward Euler.

2. Restriction to the first coarse grid, i.e., interpolate the problem to a grid that contains fewer time values, say every second or every third time value.
3. Relaxation on the first coarse grid
4. Restriction to the second coarse grid and so on...
5. When a coarse grid of trivial size (say 2 time steps) is reached, it is solved exactly.
6. The solution is then interpolated from the coarsest grid to the finest grid

One XBraid iteration is called a *cycle* and these cycles continue until the solution is accurate enough. This is depicted in the next figure, where only a few iterations are required for this simple problem.

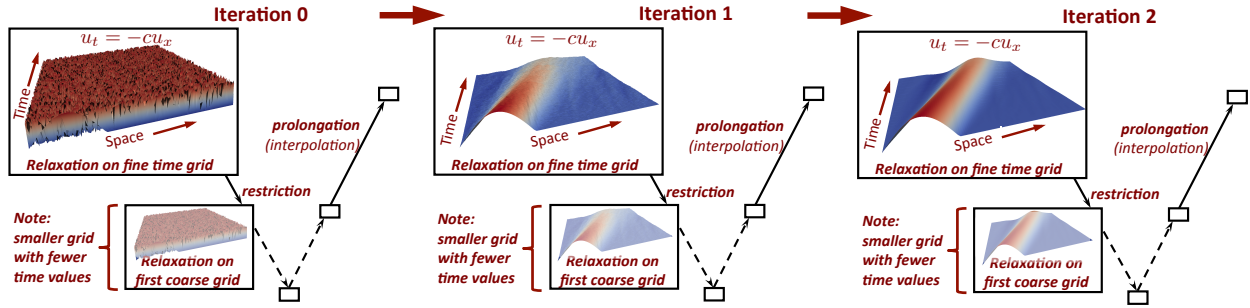


Figure 2: XBraid iterations.

There are a few important points to make.

- The coarse time grids allow for global propagation of information across space-time with only one XBraid iteration. This is visible in the above figure by observing how the solution is updated from iteration 0 to iteration 1.
- Using coarser (cheaper) grids to correct the fine grid is analogous to spatial multigrid.
- Only a few XBraid iterations are required to find the solution over 1024 time steps. Therefore if enough processors are available to parallelize XBraid, we can see a speedup over traditional time stepping (more on this later).
- This is a simple example, with evenly space time steps. XBraid is structured to handle variable time step sizes and adaptive time step sizes, and these features will be coming.

To firm up our understanding, let's do a little math. Assume that you have a general system of ordinary differential equations (ODEs),

$$u'(t) = f(t, u(t)), \quad u(0) = u_0, \quad t \in [0, T].$$

Next, let $t_i = i\delta t, i = 0, 1, \dots, N$ be a temporal mesh with spacing $\delta t = T/N$, and u_i be an approximation to $u(t_i)$. A general one-step time discretization is now given by

$$\begin{aligned} u_0 &= g_0 \\ u_i &= \Phi_i(u_{i-1}) + g_i, \quad i = 1, 2, \dots, N. \end{aligned}$$

Traditional time marching would first solve for $i = 1$, then solve for $i = 2$, and so on. For linear time propagators $\{\Phi_i\}$, this can also be expressed as applying a direct solver (a forward solve) to the following system:

$$A\mathbf{u} \equiv \begin{pmatrix} I & & & \\ -\Phi_1 & I & & \\ & \ddots & \ddots & \\ & & -\Phi_N & I \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_N \end{pmatrix} = \begin{pmatrix} g_0 \\ g_1 \\ \vdots \\ g_N \end{pmatrix} \equiv \mathbf{g}$$

or

$$A\mathbf{u} = \mathbf{g}.$$

This process is optimal and $O(N)$, but it is sequential. XBraid achieves parallelism in time by replacing this sequential solve with an optimal multigrid reduction iterative method¹ applied to only the time dimension. This approach is

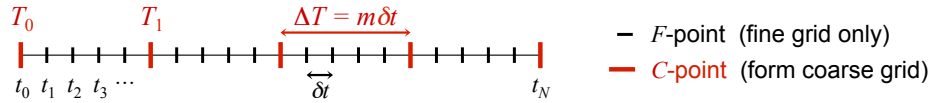
- nonintrusive, in that it coarsens only in time and the user defines Φ . Thus, users can continue using existing time stepping codes by wrapping them into our framework.
- optimal and $O(N)$, but $O(N)$ with a higher constant than time stepping. Thus with enough computational resources, XBraid will outperform sequential time stepping.
- highly parallel

We now describe the two-grid process in more detail, with the multilevel analogue being a recursive application of the process. We also assume that Φ is constant for notational simplicity. XBraid coarsens in the time dimension with factor $m > 1$ to yield a coarse time grid with $N_\Delta = N/m$ points and time step $\Delta T = m\delta t$. The corresponding coarse grid problem,

$$A_\Delta = \begin{pmatrix} I & & & \\ -\Phi_\Delta & I & & \\ & \ddots & \ddots & \\ & & -\Phi_\Delta & I \end{pmatrix},$$

is obtained by defining coarse grid propagators $\{\Phi_\Delta\}$ which are at least as cheap to apply as the fine scale propagators $\{\Phi\}$. The matrix A_Δ has fewer rows and columns than A , e.g., if we are coarsening in time by 2, A_Δ has one half as many rows and columns.

This coarse time grid induces a partition of the fine grid into C-points (associated with coarse grid points) and F-points, as visualized next. C-points exist on both the fine and coarse time grid, but F-points exist only on the fine time scale.



Every multigrid algorithm requires a relaxation method and an approach to transfer values between grids. Our relaxation scheme alternates between so-called F-relaxation and C-relaxation as illustrated next. F-relaxation updates the F-point values $\{u_j\}$ on interval (T_i, T_{i+1}) by simply propagating the C-point value u_{mi} across the interval using the time propagator $\{\Phi\}$. While this is a sequential process, each F-point interval update is independent from the others and can be computed in parallel. Similarly, C-relaxation updates the C-point value u_{mi} based on the F-point value u_{mi-1} and these updates can also be computed in parallel. This approach to relaxation can be thought of as line relaxation in space in that the residual is set to 0 for an entire time step.

The F updates are done simultaneously in parallel, as depicted next.

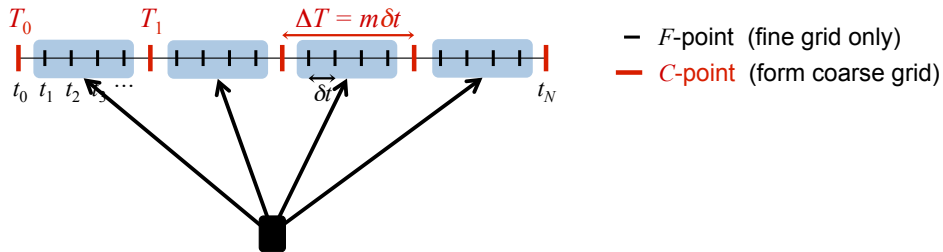
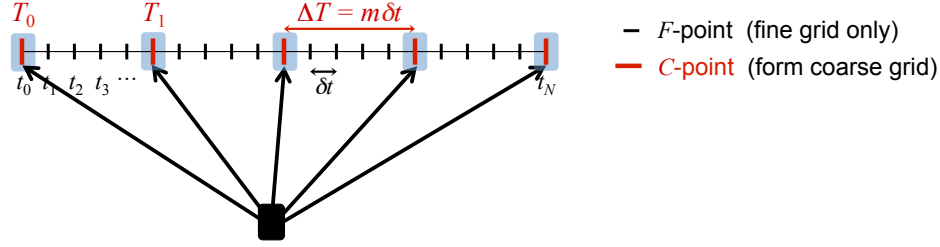


Figure 3: Update all F-point intervals in parallel, using the time propagator Φ .

Following the F sweep, the C updates are also done simultaneously in parallel, as depicted next.

¹ Ries, Manfred, Ulrich Trottenberg, and Gerd Winter. "A note on MGR methods." Linear Algebra and its Applications 49 (1983): 1-26.

Figure 4: Update all C-points in parallel, using the time propagator Φ .

In general, FCF- and F-relaxation will refer to the relaxation methods used in XBraid. We can say

- FCF- or F-relaxation is highly parallel.
- But, a sequential component exists equaling the number of F-points between two C-points.
- XBraid uses regular coarsening factors, i.e., the spacing of C-points happens every m points.

After relaxation, comes forming the coarse grid error correction. To move quantities to the coarse grid, we use the restriction operator R which simply injects values at C-points from the fine grid to the coarse grid,

$$R = \begin{pmatrix} I & & & & \\ 0 & & & & \\ \vdots & & & & \\ 0 & & I & & \\ & 0 & & & \\ & \vdots & & & \\ & 0 & & & \\ & & \ddots & & \end{pmatrix}^T.$$

The spacing between each I is $m - 1$ block rows. While injection is simple, XBraid always does an F-relaxation sweep before the application of R , which is equivalent to using the transpose of harmonic interpolation for restriction (see [Parallel Time Integration with Multigrid](#)).

To define the coarse grid equations, we apply the Full Approximation Scheme (FAS) method, which is a nonlinear version of multigrid. This is to accommodate the general case where f is a nonlinear function. In FAS, the solution guess and residual (i.e., $\mathbf{u}, \mathbf{g} - A\mathbf{u}$) are restricted. This is in contrast to linear multigrid which typically restricts only the residual equation to the coarse grid. This algorithmic change allows for the solution of general nonlinear problems. For more details, see [PDF](#) by Van Henson for a good introduction to FAS. However, FAS was originally invented by Achi Brandt.

A central question in applying FAS is how to form the coarse grid matrix A_Δ , which in turn asks how to define the coarse grid time stepper Φ_Δ . One of the simplest choices (and one frequently used in practice) is to let Φ_Δ simply be Φ but with the coarse time step size $\Delta T = m\delta t$. For example, if $\Phi = (I - \delta t A)^{-1}$ for some backward Euler scheme, then $\Phi_\Delta = (I - m\delta t A)^{-1}$ would be one choice.

With a Φ_Δ defined, the coarse grid equation

$$A_\Delta(\mathbf{v}_\Delta) = A_\Delta(\mathbf{u}_\Delta) + \mathbf{r}_\Delta$$

is then solved. Finally, FAS defines a coarse grid error approximation $\mathbf{e}_\Delta = \mathbf{v}_\Delta - \mathbf{u}_\Delta$, which is interpolated with P_Φ back to the fine grid and added to the current solution guess. Interpolation is equivalent to injecting the coarse grid to the C-points on the fine grid, followed by an F-relaxation sweep (i.e., it is equivalent to harmonic interpolation, as mentioned

above about restriction). That is,

$$P_\Phi = \begin{pmatrix} I & & & & \\ \Phi & & & & \\ \Phi^2 & & & & \\ \vdots & & & & \\ \Phi^{m-1} & & & & \\ & I & & & \\ & \Phi & & & \\ & \Phi^2 & & & \\ & \vdots & & & \\ & \Phi^{m-1} & & & \\ & & \ddots & & \end{pmatrix},$$

where m is the coarsening factor. See [Two-Grid Algorithm](#) for a concise description of the FAS algorithm for MGRIT.

2.2.1 Two-Grid Algorithm

The two-grid FAS process is captured with this algorithm. Using a recursive coarse grid solve (i.e., step 3 becomes a recursive call) makes the process multilevel. Halting is done based on a residual tolerance. If the operator is linear, this FAS cycle is equivalent to standard linear multigrid. Note that we represent A as a function below, whereas the above notation was simplified for the linear case.

1. Relax on $A(\mathbf{u}) = \mathbf{g}$ using FCF-relaxation
2. Restrict the fine grid approximation and its residual:

$$\mathbf{u}_\Delta \leftarrow R\mathbf{u}, \quad \mathbf{r}_\Delta \leftarrow R(\mathbf{g} - A(\mathbf{u})),$$

which is equivalent to updating each individual time step according to

$$u_{\Delta,i} \leftarrow u_{mi}, \quad r_{\Delta,i} \leftarrow g_{mi} - A(\mathbf{u})_{mi} \quad \text{for } i = 0, \dots, N_\Delta.$$

3. Solve $A_\Delta(\mathbf{v}_\Delta) = A_\Delta(\mathbf{u}_\Delta) + \mathbf{r}_\Delta$
4. Compute the coarse grid error approximation: $\mathbf{e}_\Delta = \mathbf{v}_\Delta - \mathbf{u}_\Delta$
5. Correct: $\mathbf{u} \leftarrow \mathbf{u} + P\mathbf{e}_\Delta$

This is equivalent to updating each individual time step by adding the error to the values of \mathbf{u} at the C-points:

$$u_{mi} = u_{mi} + e_{\Delta,i},$$

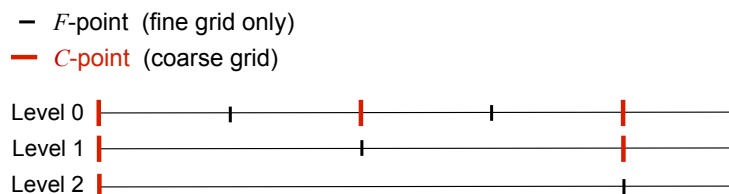
followed by an F-relaxation sweep applied to \mathbf{u} .

2.2.2 Summary

In summary, a few points are

- XBraid is an iterative solver for the global space-time problem.
- The user defines the time stepping routine Φ and can wrap existing code to accomplish this.
- XBraid convergence will depend heavily on how well Φ_Δ approximates Φ^m , that is how well a time step size of $m\delta t = \Delta T$ will approximate m applications of the same time integrator for a time step size of δt . This is a subject of research, but this approximation need not capture fine scale behavior, which is instead captured by relaxation on the fine grid.

- The coarsest grid is solved exactly, i.e., sequentially, which can be a bottleneck for two-level methods like Parareal,² but not for a multilevel scheme like XBraid where the coarsest grid is of trivial size.
- By forming the coarse grid to have the same sparsity structure and time stepper as the fine grid, the algorithm can recur easily and efficiently.
- Interpolation is ideal or exact, in that an application of interpolation leaves a zero residual at all F-points.
- The process is applied recursively until a trivially sized temporal grid is reached, e.g., 2 or 3 time points. Thus, the coarsening rate m determines how many levels there are in the hierarchy. For instance in this figure, a 3 level hierarchy is shown. Three levels are chosen because there are six time points, $m = 2$ and $m^2 < 6 \leq m^3$. If the coarsening rate had been $m = 4$ then there would only be two levels because, there would be no more points to coarsen!



By default, XBraid will subdivide the time domain into evenly sized time steps. XBraid is structured to handle variable time step sizes and adaptive time step sizes, and these features are coming.

2.3 Overview of the XBraid Code

XBraid is designed to run in conjunction with an existing application code that can be wrapped per our interface. This application code will implement some time marching simulation like fluid flow. Essentially, the user has to take their application code and extract a stand-alone time-stepping function Φ that can evolve a solution from one time value to another, regardless of time step size. After this is done, the XBraid code takes care of the parallelism in the time dimension.

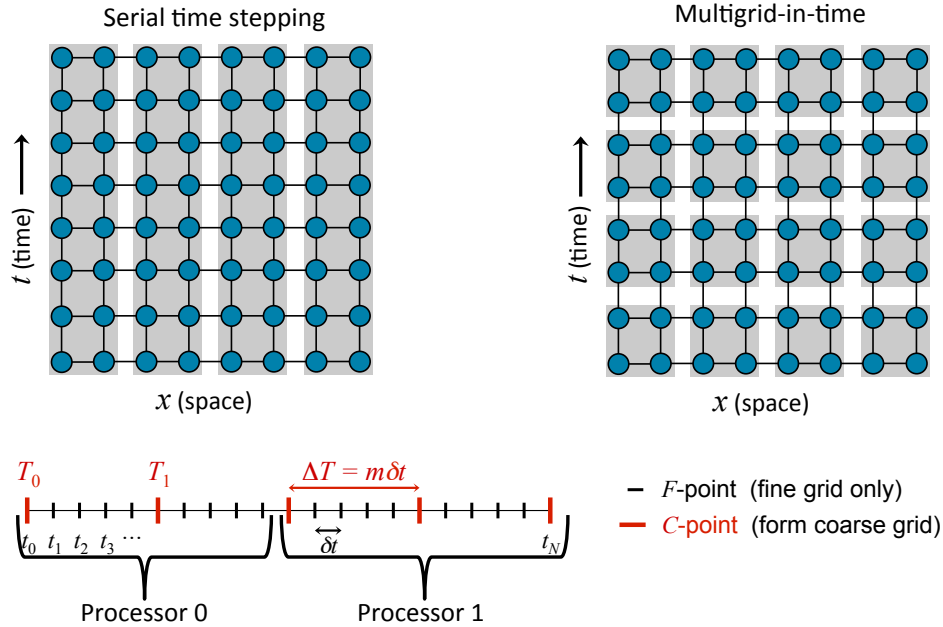
XBraid

- is written in C and can easily interface with Fortran and C++
- uses MPI for parallelism
- self documents through comments in the source code and through *.md files
- functions and structures are prefixed by *braid*
 - User routines are prefixed by `braid_`
 - Developer routines are prefixed by `_braid_`

2.3.1 Parallel decomposition and memory

- XBraid decomposes the problem in parallel as depicted next. As you can see, traditional time stepping only stores one time step at a time, but only enjoys a spatial data decomposition and spatial parallelism. On the other hand, XBraid stores multiple time steps simultaneously and each processor holds a space-time chunk reflecting both the spatial and temporal parallelism.
- XBraid only handles temporal parallelism and is agnostic to the spatial decomposition. See [braid_Split-Commworld](#). Each processor owns a certain number of CF intervals of points. In the following figure, processor 1 and processor 2 each own 2 CF intervals. XBraid distributes intervals evenly on the finest grid.

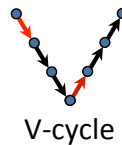
² Lions, J., Yvon Maday, and Gabriel Turinici. "A"parareal" in time discretization of PDE's." Comptes Rendus de l'Academie des Sciences Series I Mathematics 332.7 (2001): 661-668.



- XBraid increases the parallelism significantly, but now several time steps need to be stored, requiring more memory. XBraid employs two strategies to address the increased memory costs.
 - First, one need not solve the whole problem at once. Storing only one space-time slab is advisable. That is, solve for as many time steps (say k time steps) as you have available memory for. Then move on to the next k time steps.
 - Second, XBraid only stores the C -points. Whenever an F -point is needed, it is generated by F -relaxation. More precisely, we only store the red C -point time values in the previous figure. Coarsening is usually aggressive with $m = 8, 16, 32, \dots$, so the storage requirements of XBraid are significantly reduced when compared to storing all of the time values.

2.3.2 Cycling and relaxation strategies

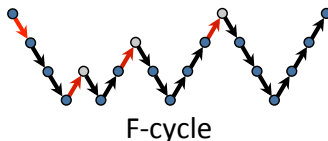
There are two main cycling strategies available in XBraid, F - and V -cycles. These two cycles differ in how often and the order in which coarse levels are visited. A V -cycle is depicted next, and is a simple recursive application of the [Two-Grid Algorithm](#).



An F -cycle visits coarse grids more frequently and in a different order. Essentially, an F -cycle uses a V -cycle as the post-smoother, which is an expensive choice for relaxation. But, this extra work gives you a closer approximation to a two-grid cycle, and a faster convergence rate at the extra expense of more work. The effectiveness of a V -cycle as a relaxation scheme can be seen in Figure 2, where one V -cycle globally propagates and *smooths* the error. The cycling strategy of an F -cycle is depicted next.

Next, we make a few points about F - versus V -cycles.

- One V -cycle iteration is cheaper than one F -cycle iteration.



- But, F-cycles often converge more quickly. For some test cases, this difference can be quite large. The cycle choice for the best time to solution will be problem dependent. See [Heat equation example](#) for a case study of cycling strategies.

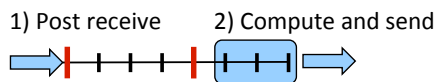
The number of FC relaxation sweeps is another important algorithmic setting. Note that at least one F-relaxation sweep is always done on a level. A few summary points about relaxation are as follows.

- Using FCF (or even FCFCF, FCFCFCF) relaxation, corresponding to passing *braid_SetNRelax* a value of 1, 2 or 3 respectively, will result in an XBraid cycle that converges more quickly as the number of relaxations grows.
- But as the number of relaxations grows, each XBraid cycle becomes more expensive. The optimal relaxation strategy for the best time to solution will be problem dependent.
- However, a good first step is to try FCF on all levels (i.e., *braid_SetNRelax(core, -1, 1)*).
- A common optimization is to first set FCF on all levels (i.e., *braid_setnrelax(core, -1, 1)*), but then overwrite the FCF option on level 0 so that only F-relaxation is done on level 0, (i.e., *braid_setnrelax(core, 0, 1)*). This strategy can work well with F-cycles.
- See [Heat equation example](#) for a case study of relaxation strategies.

Last, [Parallel Time Integration with Multigrid](#) has a more in depth case study of cycling and relaxation strategies

2.3.3 Overlapping communication and computation

XBraid effectively overlaps communication and computation. The main computational kernel of XBraid is one relaxation sweep touching all the CF intervals. At the start of a relaxation sweep, each process first posts a non-blocking receive at its left-most point. It then carries out F-relaxation in each interval, starting with the right-most interval to send the data to the neighboring process as soon as possible. If each process has multiple CF intervals at this XBraid level, the strategy allows for complete overlap.



2.3.4 Configuring the XBraid Hierarchy

Some of the more basic XBraid function calls allow you to control aspects discussed here.

- [braid_SetFMG](#): switches between using F- and V-cycles.
- [braid_SetMaxIter](#): sets the maximum number of XBraid iterations
- [braid_SetCFactor](#): sets the coarsening factor for any (or all levels)
- [braid_SetNRelax](#): sets the number of CF-relaxation sweeps for any (or all levels)
- [braid_SetRelTol](#), [braid_SetAbsTol](#): sets the stopping tolerance
- [braid_SetMinCoarse](#): sets the minimum possible coarse grid size

- [braid_SetMaxLevels](#): sets the maximum number of levels in the XBraid hierarchy

2.3.5 Halting tolerance

Another important configuration aspect regards setting a residual halting tolerance. Setting a tolerance involves these three XBraid options:

1. [braid_PtFcnSpatialNorm](#)

This user-defined function carries out a spatial norm by taking the norm of a `braid_Vector`. A common choice is the standard Euclidean norm (2-norm), but many other choices are possible, such as an L2-norm based on a finite element space.

2. [braid_SetTemporalNorm](#)

This option determines how to obtain a global space-time residual norm. That is, this decides how to combine the spatial norms returned by [braid_PtFcnSpatialNorm](#) at each time step to obtain a global norm over space and time. It is this global norm that then controls halting.

There are three options for setting the *tnorm* value passed to [braid_SetTemporalNorm](#). We let the summation index *i* be over all C-point values on the fine time grid, *k* refer to the current XBraid iteration, *r* be residual values, *space_time* norms be a norm over the entire space-time domain and *spatial_norm* be the user-defined spatial norm from [braid_PtFcnSpatialNorm](#). Thus, r_i is the residual at the *i*th C-point, and $r^{(k)}$ is the residual at the *k*th XBraid iteration. The three options are then defined as,

- *tnorm=1*: One-norm summation of spatial norms

$$\|r^{(k)}\|_{\text{space_time}} = \sum_i \|r_i^{(k)}\|_{\text{spatial_norm}}$$

If [braid_PtFcnSpatialNorm](#) is the one-norm over space, then this is equivalent to the one-norm of the global space-time residual vector.

- *tnorm=2*: Two-norm summation of spatial norms

$$\|r^{(k)}\|_{\text{space_time}} = \left(\sum_i \|r_i^{(k)}\|_{\text{spatial_norm}}^2 \right)^{1/2}$$

If [braid_PtFcnSpatialNorm](#) is the Euclidean norm (two-norm) over space, then this is equivalent to the Euclidean-norm of the global space-time residual vector.

- *tnorm=3*: Infinity-norm combination of spatial norms

$$\|r^{(k)}\|_{\text{space_time}} = \max_i \|r_i^{(k)}\|_{\text{spatial_norm}}$$

If [braid_PtFcnSpatialNorm](#) is the infinity-norm over space, then this is equivalent to the infinity-norm of the global space-time residual vector.

The default choice is *tnorm=2*

3. [braid_SetAbsTol](#), [braid_SetRelTol](#)

- If an absolute tolerance is used, then

$$\|r^{(k)}\|_{\text{space_time}} < \text{tol}$$

defines when to halt.

- If a relative tolerance is used, then

$$\frac{\|r^{(k)}\|_{\text{space_time}}}{\|r^{(0)}\|_{\text{space_time}}} < \text{tol}$$

defines when to halt. That is, the current *k*th residual is scaled by the initial residual before comparison to the halting tolerance. This is similar to typical relative residual halting tolerances used in spatial multigrid, but can be a dangerous choice in this setting.

Care should be practiced when choosing a halting tolerance. For instance, if a relative tolerance is used, then issues can arise when the initial guess is zero for large numbers of time steps. Taking the case where the initial guess (defined by `braid_PtFcnInit`) is 0 for all time values $t > 0$, the initial residual norm will essentially only be nonzero at the first time value,

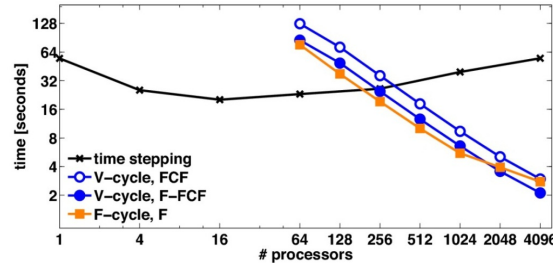
$$\|r^{(0)}\|_{\text{space_time}} \approx \|r_1^{(k)}\|_{\text{spatial_norm}}$$

This will skew the relative halting tolerance, especially if the number of time steps increases, but the initial residual norm does not.

A better strategy is to choose an absolute tolerance that takes your space-time domain size into account, as in Section [Heat equation example](#), or to use an infinity-norm temporal norm option.

2.3.6 Heat equation example

Here is some experimental data for the 2D heat equation, $u_t = u_{xx} + u_{yy}$ generated by examples/drive-02. The problem



setup is as follows.

- Backwards Euler is used as the time stepper.
- We used a Linux cluster with 4 cores per node, a Sandybridge Intel chipset, and a fast Infiniband interconnect.
- The space-time problem size was $129^2 \times 16,192$ over the unit cube $[0, 1] \times [0, 1] \times [0, 1]$.
- The coarsening factor was $m = 16$ on the finest level and $m = 2$ on coarser levels.
- Since 16 processors optimized the serial time stepping approach, 16 processors in space are also used for the XBraid experiments. So for instance 512 processors in the plot corresponds to 16 processors in space and 32 processors in time, $16 * 32 = 512$. Thus, each processor owns a space-time hypercube of $(129^2/16) \times (16, 192/32)$. See [Parallel decomposition and memory](#) for a depiction of how XBraid breaks the problem up.
- Various relaxation and V and F cycling strategies are experimented with.
 - *V-cycle, FCF* denotes V-cycles and FCF-relaxation on each level.
 - *V-cycle, F-FCF* denotes V-cycles and F-relaxation on the finest level and FCF-relaxation on all coarser levels.
 - *F-cycle, F* denotes F-cycles and F-relaxation on each level.
- The initial guess at time values for $t > 0$ is zero, which is typical.
- The halting tolerance corresponds to a discrete L2-norm and was

$$\text{tol} = \frac{10^{-8}}{\sqrt{(h_x)^2 h_t}},$$

where h_x and h_t are the spatial and temporal grid spacings, respectively.

This corresponds to passing `tol` to `braid_SetAbsTol`, passing 2 to `braid_SetTemporalNorm` and defining `braid_PtFcnSpatialNorm` to be the standard Euclidean 2-norm. All together, this appropriately scales the space-time residual in way that is relative to the number of space-time grid points (i.e., it approximates the L2-norm).

Regarding the performance, we can say

- The best speedup is 10x and this would grow if more processors were available.
- Although not shown, the iteration counts here are about 10-15 XBraid iterations. See [Parallel Time Integration with Multigrid](#) for the exact iteration counts.
- At smaller core counts, serial time stepping is faster. But at about 256 processors, there is a crossover and XBraid is faster.
- You can see the impact of the cycling and relaxation strategies discussed in [Cycling and relaxation strategies](#). For instance, even though *V-cycle, F-FCF* is a weaker relaxation strategy than *V-cycle, FCF* (i.e., the XBraid convergence is slower), *V-cycle, F-FCF* has a faster time to solution than *V-cycle, FCF* because each cycle is cheaper.
- In general, one level of aggressive coarsening (here by a factor 16) followed by slower coarsening was found to be best on this machine.

Achieving the best speedup can require some tuning, and it is recommended to read [Parallel Time Integration with Multigrid](#) where this 2D heat equation example is explored in much more detail.

2.4 Citing XBraid

To cite XBraid, please state in your text the version number from the VERSION file, and please cite the project website in your bibliography as

[1] XBraid: Parallel multigrid in time. <http://llnl.gov/casc/xbraid>.

The corresponding BibTex entry is

```
@misc{xbraid-package,
  title = {{XB}raid: Parallel multigrid in time},
  howpublished = {\url{http://llnl.gov/casc/xbraid}}
}
```

2.5 Summary

- XBraid applies multigrid to the time dimension.
 - This exposes concurrency in the time dimension.
 - The potential for speedup is large, 10x, 100x, ...
- This is a non-intrusive approach, with an unchanged time discretization defined by user.
- Parallel time integration is only useful beyond some scale. This is evidenced by the experimental results below. For smaller numbers of cores sequential time stepping is faster, but at larger core counts XBraid is much faster.
- The more time steps that you can parallelize over, the better your speedup will be.
- XBraid is optimal for a variety of parabolic problems (see the examples directory).

3 Example

A Simple Example

User Defined Structures and Wrappers

As mentioned, the user must wrap their existing time stepping routine per the XBraid interface. To do this, the user must define two data structures and some wrapper routines. To make the idea more concrete, we now give these function definitions from examples/drive-01, which implements a scalar ODE, $u_t = \lambda u$.

The two data structures are:

1. **App:** This holds a wide variety of information and is *global* in that it is passed to every function. This structure holds everything that the user will need to carry out a simulation. Here, this is just the global MPI communicator and few values describing the temporal domain.

```
typedef struct _braid_App_struct
{
    MPI_Comm    comm;
    double      tstart;
    double      tstop;
    int         ntime;
} my_App;
```

2. **Vector:** this defines (roughly) a state vector at a certain time value. It could also contain any other information related to this vector which is needed to evolve the vector to the next time value, like mesh information. Here, the vector is just a scalar double.

```
typedef struct _braid_Vector_struct
{
    double value;
} my_Vector;
```

The user must also define a few wrapper routines. Note, that the app structure is the first argument to every function.

1. **Phi:** This function tells XBraid how to take a time step, and is the core user routine. The user must advance the vector u from time $tstart$ to time $tstop$. Note how the time values are given to the user through the *status* structure and associated Get routines. The *rfactor_ptr* parameter is an advanced topic not used here.

Here advancing the solution just involves the scalar λ .

Importantly, the g_i function (from [Overview of the XBraid Algorithm](#)) must be incorporated into Phi, so that $\Phi(u_i) \rightarrow u_{i+1}$

```
int
my_Phi(braid_App      app,
       braid_Vector   u,
       braid_PhiStatus status)
{
    double tstart;          /* current time */
    double tstop;           /* evolve to this time*/
    braid_PhiStatusGetTstartTstop(status, &tstart, &tstop);

    /* On the finest grid, each value is half the previous value */
    (u->value) = pow(0.5, tstop-tstart)*(u->value);

    /* Zero rhs for now */
    (u->value) += 0.0;

    /* no refinement */
    braid_PhiStatusSetRFactor(status, 1);

    return 0;
}
```

2. **Init:** This function tells XBraid how to initialize a vector at time t . Here that is just allocating and setting a scalar on the heap.

```
int
my_Init(braid_App      app,
        double         t,
        braid_Vector *u_ptr)
{
    my_Vector *u;
```

```

    u = (my_Vector *) malloc(sizeof(my_Vector));
    if (t == 0.0)
    {
        /* Initial guess */
        (u->value) = 1.0;
    }
    else
    {
        /* Random between 0 and 1 */
        (u->value) = ((double)rand()) / RAND_MAX;
    }
    *u_ptr = u;

    return 0;
}

```

3. **Clone:** This function tells XBraid how to clone a vector into a new vector.

```

int
my_Clone(braid_App app,
        braid_Vector u,
        braid_Vector *v_ptr)
{
    my_Vector *v;

    v = (my_Vector *) malloc(sizeof(my_Vector));
    (v->value) = (u->value);
    *v_ptr = v;

    return 0;
}

```

4. **Free:** This function tells XBraid how to free a vector.

```

int
my_Free(braid_App app,
        braid_Vector u)
{
    free(u);

    return 0;
}

```

5. **Sum:** This function tells XBraid how to sum two vectors (AXPY operation).

```

int
my_Sum(braid_App app,
       double alpha,
       braid_Vector x,
       double beta,
       braid_Vector y)
{
    (y->value) = alpha*(x->value) + beta*(y->value);

    return 0;
}

```

6. **SpatialNorm:** This function tells XBraid how to take the norm of a braid_Vector and is used for halting. This norm is only over space. A common norm choice is the standard Euclidean norm, but many other choices are possible, such as an L2-norm based on a finite element space. The norm choice should be based on what makes sense for you problem. How to accumulate spatial norm values to obtain a global space-time residual norm for halting decisions is controlled by [braid_SetTemporalNorm](#).

```

int
my_SpatialNorm(braid_App app,
               braid_Vector u,
               double *norm_ptr)
{
    double dot;

    dot = (u->value)*(u->value);
    *norm_ptr = sqrt(dot);
}

```

```

    return 0;
}

```

7. **Access:** This function allows the user access to XBraid and the current solution vector at time t . This is most commonly used to print solution(s) to screen, file, etc... The user defines what is appropriate output. Notice how you are told the time value t of the vector u and even more information in *astatus*. This lets you tailor the output to only certain time values at certain XBraid iterations. Querying *astatus* for such information is done through *braid_AccessStatusGet**(..)* routines.

The frequency of the calls to *access* is controlled through *braid_SetAccessLevel*. For instance, if *access_level* is set to 2, then *access* is called every XBraid iteration and on every XBraid level. In this case, querying *astatus* to determine the current XBraid level and iteration will be useful. This scenario allows for even more detailed tracking of the simulation.

Eventually, this routine will allow for broader access to XBraid and computational steering.

See examples/drive-02 and examples/drive-04 for more advanced uses of the *access* function. Drive-04 uses *access* to write solution vectors to a GLVIS visualization port, and examples/drive-02 uses *access* to write to .vtu files.

```

int
my_Access(braid_App      app,
          braid_Vector    u,
          braid_AccessStatus astatus)
{
    MPI_Comm    comm    = (app->comm);
    double      tstart  = (app->tstart);
    double      tstop   = (app->tstop);
    int         ntime   = (app->ntime);
    int         index, myid;
    char        filename[255];
    FILE        *file;
    double      t;

    braid_AccessStatusGetT(astatus, &t);
    index = ((t-tstart) / ((tstop-tstart)/ntime) + 0.1);

    MPI_Comm_rank(comm, &myid);

    sprintf(filename, "%s.%07d.%05d", "drive-01.out", index, myid);
    file = fopen(filename, "w");
    fprintf(file, "%.14e\n", (u->value));
    fflush(file);
    fclose(file);

    return 0;
}

```

8. **BufSize, BufPack, BufUnpack:** These three routines tell XBraid how to communicate vectors between processors. *BufPack* packs a vector into a `void *` buffer for MPI and then *BufUnPack* unpacks it from `void *` to vector. Here doing that for a scalar is trivial. *BufSize* computes the upper bound for the size of an arbitrary vector.

Note how *BufPack* also returns a size pointer. This size pointer should be the exact number of bytes packed, while *BufSize* should provide only an upper-bound on a possible buffer size. This flexibility allows for variable spatial grid sizes to result in smaller messages sent when appropriate. **To avoid MPI issues, it is very important that BufSize be pessimistic, provide an upper bound, and return the same value across processors.**

In general, the buffer should be self-contained. The receiving processor should be able to pull all necessary information from the buffer in order to properly interpret and unpack the buffer.

```

int
my_BufSize(braid_App app,
           int      *size_ptr)

```

```

{
    *size_ptr = sizeof(double);
    return 0;
}

int
my_BufPack(braid_App    app,
           braid_Vector u,
           void          *buffer,
           braid_Int     *size_ptr)
{
    double *dbuffer = buffer;

    dbuffer[0] = (u->value);
    *size_ptr = sizeof(double);

    return 0;
}

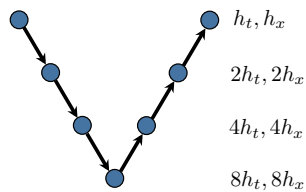
int
my_BufUnpack(braid_App    app,
              void          *buffer,
              braid_Vector *u_ptr)
{
    double *dbuffer = buffer;
    my_Vector *u;

    u = (my_Vector *) malloc(sizeof(my_Vector));
    (u->value) = dbuffer[0];
    *u_ptr = u;

    return 0;
}

```

9. **Coarsen, Restrict** (optional): These are advanced options that allow for coarsening in space while you coarsen in time. This is useful for maintaining stable explicit schemes on coarse time scales and is not needed here. See for instance examples/drive-04 and examples/drive-05 which use these routines. These functions allow you vary the spatial mesh size on XBraid levels as depicted here where the spatial and temporal grid sizes are halved every level.



10. Adaptive and variable time stepping is in the works to be implemented. The *rfactor* parameter in *Phi* will allow this.

Running XBraid

A typical flow of events in the `main` function is to first initialize the `app` structure.

```

/* set up app structure */
app = (my_App *) malloc(sizeof(my_App));
(app->comm) = comm;
(app->tstart) = tstart;
(app->tstop) = tstop;
(app->ntime) = ntime;

```

Then, the data structure definitions and wrapper routines are passed to XBraid. The core structure is used by XBraid for internal data structures.

```
braid_Core core;
braid_Init(MPI_COMM_WORLD, comm, tstart, tstop, ntime, app,
           my_Phi, my_Init, my_Clone, my_Free, my_Sum, my_SpatialNorm,
           my_Access, my_BufSize, my_BufPack, my_BufUnpack, &core);
```

Then, XBraid options are set.

```
braid_SetPrintLevel( core, 1);
braid_SetMaxLevels(core, max_levels);
braid_SetNRelax(core, -1, nrelax);
braid_SetAbsTol(core, tol);
braid_SetCFactor(core, -1, cfactor);
braid_SetMaxIter(core, max_iter);
```

Then, the simulation is run.

```
braid_Drive(core);
```

Then, we clean up.

```
braid_Destroy(core);
```

Finally, to run drive-01, type

```
drive-01 -ml 5
```

This will run drive-01. See examples/drive-0* for more extensive examples.

Testing XBraid

The best overall test for XBraid, is to set the maximum number of levels to 1 (see [braid_SetMaxLevels](#)) which will carry out a sequential time stepping test. Take the output given to you by your *Access* function and compare it to output from a non-XBraid run. Is everything OK? Once this is complete, repeat for multilevel XBraid, and check that the solution is correct (that is, it matches a serial run to within tolerance).

At a lower level, to do sanity checks of your data structures and wrapper routines, there are also XBraid test functions, which can be easily run. The test routines also take as arguments the app structure, spatial communicator *comm_x*, a stream like *stdout* for test output and a time step size to test *dt*. After these arguments, function pointers to wrapper routines are the rest of the arguments. Some of the tests can return a boolean variable to indicate correctness.

```
/* Test init(), access(), free() */
braid_TestInitAccess( app, comm_x, stdout, dt, my_Init, my_Access, my_Free);

/* Test clone() */
braid_TestClone( app, comm_x, stdout, dt, my_Init, my_Access, my_Free, my_Clone);

/* Test sum() */
braid_TestSum( app, comm_x, stdout, dt, my_Init, my_Access, my_Free, my_Clone, my_Sum);

/* Test spatialnorm() */
correct = braid_TestSpatialNorm( app, comm_x, stdout, dt, my_Init, my_Free, my_Clone,
                                my_Sum, my_SpatialNorm);

/* Test bufsize(), bufpack(), bufunpack() */
correct = braid_TestBuf( app, comm_x, stdout, dt, my_Init, my_Free, my_Sum, my_SpatialNorm,
                        my_BufSize, my_BufPack, my_BufUnpack);

/* Test coarsen and refine */
correct = braid_TestCoarsenRefine(app, comm_x, stdout, 0.0, dt, 2*dt, my_Init,
                                my_Access, my_Free, my_Clone, my_Sum, my_SpatialNorm,
                                my_CoarsenInjection, my_Refine);
correct = braid_TestCoarsenRefine(app, comm_x, stdout, 0.0, dt, 2*dt, my_Init,
                                my_Access, my_Free, my_Clone, my_Sum, my_SpatialNorm,
                                my_CoarsenBilinear, my_Refine);
```

4 Building XBraid

- Copyright information and licensing restrictions can be found in the files COPYRIGHT and LICENSE.
- To specify the compilers, flags and options for your machine, edit makefile.inc. For now, we keep it simple and avoid using configure or cmake.
- To make the library, libbraid.a,

```
$ make
```
- To make the examples

```
$ make all
```
- The makefile lets you pass some parameters like debug with

```
$ make debug=yes
```

or

```
$ make all debug=yes
```

It would also be easy to add additional parameters, e.g., to compile with insure.
- To set compilers and library locations, look in makefile.inc where you can set up an option for your machine to define simple stuff like

```
CC = mpicc
MPICC = mpicc
MPICXX = mpiCC
LFLAGS = -lm
```

5 Compiling and running the examples

Type

```
drive-0* -help
```

for instructions on how to run any driver.

To run the examples, type

```
mpirun -np 4 drive-* [args]
```

1. drive-01 is the simplest example. It implements a scalar ODE and can be compiled and run with no outside dependencies.
2. drive-02 implements the 2D heat equation on a regular grid. You must have **hypre** installed and these variables in examples/Makefile set correctly

```
HYPRE_DIR = ../../linear_solvers/hypre
HYPRE_FLAGS = -I$(HYPRE_DIR)/include
HYPRE_LIB = -L$(HYPRE_DIR)/lib -lhypre
```
3. drive-03 implements the 3D heat equation on a regular grid, and assumes **hypre** is installed just like drive-02.
4. drive-05 implements the 2D heat equation on a regular grid, but it uses spatial coarsening. This allows you to use explicit time stepping on each Braid level, regardless of time step size. It assumes **hypre** is installed just like drive-02.
5. drive-04 is a sophisticated test bed for various PDEs, mostly parabolic. It relies on the **mfem** package to create general finite element discretizations for the spatial problem. Other packages must be installed in this order.
 - Unpack and install **Metis**
 - Unpack and install **hypre**
 - Unpack and install **mfem**. Make the serial version of mfem first by only typing `make`. Then make sure to set these variables correctly in the mfem Makefile:

```
USE_METIS_5 = YES
HYPRE_DIR = where_ever_linear_solvers_is/hypre
```
 - Make **GLVIS**, which needs serial mfem. Set these variables in the glvis makefile

```
MFEM_DIR    = mfem_location
MFEM_LIB    = -L$(MFEM_DIR) -lmfem
```

- Go back to the mfem directory and type


```
make clean
make parallel
```
- Go to braid/examples and set these Makefile variables,


```
METIS_DIR = ../../metis-5.1.0/lib
MFEM_DIR = ../../mfem
MFEM_FLAGS = -I$(MFEM_DIR)
MFEM_LIB = -L$(MFEM_DIR) -lmfem -L$(METIS_DIR) -lmetis
```

 then type


```
make drive-04
```
- To run drive-04 and glvis, open two windows. In one, start a glvis session


```
./glvis
```

 Then, in the other window, run drive-04


```
mpirun -np ... drive-04 [args]
```

 Glvis will listen on a port to which drive-04 will dump visualization information.

6 Coding Style

Code should follow the `ellemtel` style. See `braid/misc/sample_c_code.c`, and for emacs and vim style files, see `braid/misc/sample.vimrc`, and `braid/misc/sample.emacs`.

7 Using Doxygen

To build the documentation, doxygen must be version 1.8 or greater. XBraid documentation uses a `markdown` syntax both in source file comments and in `*.md` files.

To make the documentation,

```
$ make user_manual
$ acread user_manual.pdf
```

or to make a more extensive reference manual for developers,

```
$ make developer_manual
$ acread developer_manual.pdf
```

Developers can run doxygen from a precompiled binary, which may or may not work for your machine,

```
/usr/casc/hypre/braid/share/doxygen/bin/doxygen
```

or build doxygen from

```
/usr/casc/hypre/braid/share/doxygen.tgz
```

- Compiling doxygen requires a number of dependencies like Bison, GraphViz and Flex. Configure will tell you what you're missing
- Unpack `doxygen.tgz`, then from the doxygen directory


```
./configure --prefix some_dir_in_your_path
make
make install
```

Documentation Strategy

- The doxygen comments are to be placed in the header files.
- A sample function declaration using the documentation approach using markdown (including typesetting equations) is in [braid.h](#) for the function [braid_Init\(\)](#)
- A sample structure documentation is in [_braid.h](#) for [_braid_Core_struct](#)
- Descriptors for files can also be added, as at the top of [braid.h](#)
- The Doxygen manual is at <http://www.stack.nl/~dimitri/doxygen/manual/index.html>

XBraid Doxygen details

The user and developer manuals are ultimately produced by Latex. The formatting of the manuals is configured according to the following.

- docs/local_doxygen.sty
 - Latex style file used
- docs/user_manual_header.tex
 - User manual title page and header info
- docs/developer_manual_header.tex
 - Developer manual title page and header info
- *.md
 - Any file ending in .md is extra documentation in markdown format, like [Introduction.md](#) or the various [Readme.md](#) files in each directory. This material can be read in plain-text or when it's compiled by Doxygen and Latex.
- docs/user_manual.conf
 - Doxygen configure file for the user manual
 - The FILE_NAMES tag is a filter to only include the user interface routines in [braid.h](#)
 - The INPUT tag orders the processing of the files and hence the section ordering
- docs/reference_manual.conf
 - Same as user_manual.conf, but the FILE_NAMES tag does not exclude any file from processing.
- docs/img
 - Contains the images
- To regenerate generic doxygen latex files, type


```
$ doxygen -w latex header.tex footer.tex doxygen.sty doxy.conf
```

If this is done, then the .conf file must be changed to use the new header file and to copy the local_doxygen.sty file to the latex directory.

8 Regression Testing

Overview

- There are three levels in the testing framework. At each level, the fine-grain output from a `testscript.sh` is dumped into a directory `testscript.dir`, with the standard out and error stored in `testscript.out` and `testscript.err`. The test `testscript.sh` passes if `testscript.err` is empty (nothing is written to standard error).
- Basic instructions: run a test with a command like


```
$ ./test.sh diffusion2D.sh
```

Then, see if `diffusion2D.err` is of size 0. If it is not, look at it's contents to see which test failed.

- To add a new regression test, create a new lowest level script like `diffusion2D.sh` and then call it from a machine script at level 2.
- Regression tests should be run before pushing code. It is recommended to run the basic (lowest level) tests like `diffusion2d.sh` or machine test like `machine-tux.sh`

Lowest Level Test Scripts

As an example, here we look at one of the lowest level tests, the `diffusion2d` test.

Files used:

- `test.sh`
- `diffusion2D.sh`
- `diffusion2D.saved`

Output:

- `diffusion2D.dir`
- `diffusion2D.err`
- `diffusion2D.out`

At this level, we execute

```
$ ./test.sh diffusion2D.sh
```

or just

```
$ ./diffusion2D.sh
```

The script `diffusion2D.sh` must create `diffusion2D.dir` and place all fine-grain test output in this directory. `test.sh` captures the standard out and error in `diffusion2D.out` and `diffusion2D.err`. The test `diffusion2D.sh` passes if `diffusion2D.err` is empty (nothing is written to standard error).

The strategy for low level scripts like `diffusion2D.sh` is to run a sequence of test drivers such as

```
$ mpirun -np 1 ../examples/drive-02 -pgrid 1 1 1 -nt 256
$ mpirun -np 4 ../examples/drive-02 -pgrid 1 1 4 -nt 256
```

The output from the first `mpirun` test must then be written to files named

```
diffusion2D.dir/unfiltered.std.out.0
diffusion2D.dir/std.out.0
diffusion2D.dir/std.err.0
```

and the second `mpirun` test similarly writes the files

```
diffusion2D.dir/unfiltered.std.out.1
diffusion2D.dir/std.out.1
diffusion2D.dir/std.err.1
```

Subsequent tests are written to higher numbered files. The `unfiltered.std.out.num` file contains all of the standard out for the test, while `std.out.num` contains filtered output (usually from a `grep` command) and could

contain the output lines such as iteration numbers and number of levels. The file `std.err.num` contains the standard error output.

To see if a test ran correctly, `std.out.num` is compared to saved output in `diffusion2D.saved`. The file `diffusion2D.saved` contains the concatenated output from all the tests that `diffusion2D.sh` will run. For the above example, this file could look like

```
# Begin Test 1
number of levels      = 6
iterations            = 16
# Begin Test 2
number of levels      = 4
iterations            = 8
```

This saved output is split into an individual file for each test (using `# Begin Test` as a delimiter) and these new files are placed in `diffusion2D.dir`. So, after running these two regression tests, `diffusion2D.dir` will contain

```
diffusion2D.saved.0
diffusion2D.saved.1
unfiltered.std.out.0
std.out.0
std.err.0
unfiltered.std.out.1
std.out.1
std.err.1
```

An individual test has passed if `std.err.num` is empty. The file `std.err.num` contains a diff between `diffusion2D.save.num` and `std.out.num` (the diff ignores whitespace and the delimiter `# Begin Test`).

Last in the directory where you ran `./test.sh diffusion2d.sh`, the files

```
diffusion2D.err
diffusion2D.out
```

will be created. If all the tests passed then `diffusion2D.err` will be empty. Otherwise, it will contain the filenames of the `std.err.num` files that are non-empty, representing failed tests.

Level 2 Scripts

As an example, here we look at one of the Level 2 tests, the machine-tux test that Jacob runs.

Files used:

- `machine-tux.sh`

Output:

- `machine-tux.dir`
- `machine-tux.err` (only generated if `autotest.sh` is used to run `machine-tux.sh`)
- `machine-tux.out` (only generated if `autotest.sh` is used to run `machine-tux.sh`)

At this level, we execute

```
./machine-tux.sh
```

The autotest framework (`autotest.sh`) calls machine scripts in this way. Each machine script should be short and call lower-level scripts like `diffusion2D.sh`. The output from lower-level scripts must be moved to `machine-tux.dir` like this:

```
$ ./test.sh diffusion2D.sh
$ mv -f diffusion2D.dir machine-tux.dir
$ mv -f diffusion2D.out machine-tux.dir
$ mv -f diffusion2D.err machine-tux.dir
```

All error files from `diffusion2D.sh` will be placed in `machine-tux.dir`, so if `machine-tux.dir` has all zero `*.err` files, then the `machine-tux` test has passed.

To begin testing on a new machine, like `vulcan`, add a new machine script similar to `machine-tux.sh` and change `autotest.sh` to recognize and run the new machine test. To then use `autotest.sh` with the machine script, you'll have to set up a passwordless connection from the new machine to

```
/usr/casc/hypre/braid/testing
```

Level 3 Script

Here we look at the highest level, where `autotest.sh` runs all of the level 2 machine tests and emails out the results.

Files used:

- `autotest.sh`

Output:

- `test/autotest_finished`
- `/usr/casc/hypre/braid/testing/AUTOTEST-20**.**.**-Day`
- Email to recipients listed in `autotest.sh`

At the highest level sits `autotest.sh` and is called automatically as a cronjob. If you just want to check to see if you've broken anything with a commit, just use lower level scripts.

There are four steps to running `autotest`.

- Step 1


```
$ ./autotest.sh -init
```

 will do a pull from master for the current working repository and recompile Braid. The `autotest` output files (`autotest.err` and `autotest.out`) and the output directory (`autotest_finished`) are initialized.
- Step 2


```
$ ./autotest.sh -tux343
```

 will run the autotests on `tux343`. This command will look for a `machine-tux.sh`, and execute it, moving the resulting


```
machine-tux.dir
machine-tux.err
machine-tux.out
```

 into `test/autotest_finished`.
- Step 3


```
$ ./autotest.sh -remote-copy
```

 will copy `/test/autotest_finished/*` to a time-stamped directory such as `/usr/casc/hypre/braid/testing/AUTOTEST-2013.11.18-Mon`. Alternatively,


```
$ ./autotest.sh -remote-copy tux343
```

 will ssh through `tux343` to copy to `/usr/casc`. Multiple machines may independently be running regression tests and then copy to `AUTOTEST-2013.11.18-Mon`.
- Step 4


```
$ ./autotest.sh -summary-email
```

 will email everyone listed in the `$email_list` (an `autotest.sh` variable)

Cronfile

To add entries to your crontab, First, put your new cronjob lines into `cronfile`. Then see what you already have in your crontab file with

```
$ crontab -l
```

Next, append to cronfile whatever you already have

```
$ crontab -l >> cronfile
```

Finally, tell crontab to use your cronfile

```
$ crontab cronfile
```

Then make sure it took affect with

```
$ crontab -l
```

Crontab entry format uses '*' to mean "every" and '*/m' to mean "every m-th". The first five entries on each line correspond respectively to:

- minute (0-56)
- hour (0-23)
- day of month (1-31)
- month (1-12)
- day of week (0-6)(0=Sunday)

Jacob's crontab (on tux343):

```
00 01 * * * source /etc/profile; source $HOME/.bashrc; cd $HOME/joint_repos/braid/test; ./autotest.sh -init
10 01 * * * source /etc/profile; source $HOME/.bashrc; cd $HOME/joint_repos/braid/test; ./autotest.sh -tux343
40 01 * * * source /etc/profile; source $HOME/.bashrc; cd $HOME/joint_repos/braid/test; ./autotest.sh -remote-c
50 01 * * * source /etc/profile; source $HOME/.bashrc; cd $HOME/joint_repos/braid/test; ./autotest.sh -summary-
00 02 * * * source /etc/profile; source $HOME/.bashrc; cd $HOME/joint_repos/braid/test; ./autotest.sh -create-t
```

9 Module Index

9.1 Modules

Here is a list of all modules:

User-written routines	25
User interface routines	29
General Interface routines	30
XBraid status routines	38
XBraid test routines	46

10 Data Structure Index

10.1 Data Structures

Here are the data structures with brief descriptions:

_braid_AccessStatus	51
_braid_AccuracyHandle	51
_braid_CoarsenRefStatus	52
_braid_CommHandle	53
_braid_Core	54
_braid_Grid	58
_braid_PhiStatus	61

11 File Index

11.1 File List

Here is a list of all files with brief descriptions:

_braid.h	Define headers for developer routines	61
braid.h	Define headers for user interface routines	66
braid_defs.h	Definitions of types, error flags, etc..	68
braid_status.h	Define headers for XBraid status structures, status get/set routines and status create/destroy routines	69
braid_test.h	Define headers for XBraid test routines	70
util.h	Define headers for utility routines	70

12 Module Documentation

12.1 User-written routines

Typedefs

- typedef struct _braid_App_struct * [braid_App](#)

- typedef struct
_braid_Vector_struct * [braid_Vector](#)
- typedef [braid_Int](#)(* [braid_PtFcnPhi](#))(braid_App app, [braid_Vector](#) u, [braid_PhiStatus](#) status)
- typedef [braid_Int](#)(* [braid_PtFcnInit](#))(braid_App app, [braid_Real](#) t, [braid_Vector](#) *u_ptr)
- typedef [braid_Int](#)(* [braid_PtFcnClone](#))(braid_App app, [braid_Vector](#) u, [braid_Vector](#) *v_ptr)
- typedef [braid_Int](#)(* [braid_PtFcnFree](#))(braid_App app, [braid_Vector](#) u)
- typedef [braid_Int](#)(* [braid_PtFcnSum](#))(braid_App app, [braid_Real](#) alpha, [braid_Vector](#) x, [braid_Real](#) beta, [braid_Vector](#) y)
- typedef [braid_Int](#)(* [braid_PtFcnSpatialNorm](#))(braid_App app, [braid_Vector](#) u, [braid_Real](#) *norm_ptr)
- typedef [braid_Int](#)(* [braid_PtFcnAccess](#))(braid_App app, [braid_Vector](#) u, [braid_AccessStatus](#) status)
- typedef [braid_Int](#)(* [braid_PtFcnBufSize](#))(braid_App app, [braid_Int](#) *size_ptr)
- typedef [braid_Int](#)(* [braid_PtFcnBufPack](#))(braid_App app, [braid_Vector](#) u, void *buffer, [braid_Int](#) *size_ptr)
- typedef [braid_Int](#)(* [braid_PtFcnBufUnpack](#))(braid_App app, void *buffer, [braid_Vector](#) *u_ptr)
- typedef [braid_Int](#)(* [braid_PtFcnCoarsen](#))(braid_App app, [braid_Vector](#) fu, [braid_Vector](#) *cu_ptr, [braid_Coarsen-RefStatus](#) status)
- typedef [braid_Int](#)(* [braid_PtFcnRefine](#))(braid_App app, [braid_Vector](#) cu, [braid_Vector](#) *fu_ptr, [braid_Coarsen-RefStatus](#) status)

12.1.1 Detailed Description

These are all user-written data structures and routines. There are two data structures ([braid_App](#) and [braid_Vector](#)) for the user to define. And, there are a variety of function interfaces (defined through function pointer declarations) that the user must implement.

12.1.2 Typedef Documentation

12.1.2.1 typedef struct _braid_App_struct* [braid_App](#)

This holds a wide variety of information and is `global` in that it is passed to every function. This structure holds everything that the user will need to carry out a simulation. For a simple example, this could just hold the global MPI communicator and a few values describing the temporal domain.

12.1.2.2 typedef [braid_Int](#)(* [braid_PtFcnAccess](#))(braid_App app, [braid_Vector](#) u, [braid_AccessStatus](#) status)

Gives user access to XBraid and to the current vector u at time t . Most commonly, this lets the user write the vector to screen, file, etc... The user decides what is appropriate. Note how you are told the time value t of the vector u and other information in *status*. This lets you tailor the output, e.g., for only certain time values at certain XBraid iterations. Querying status for such information is done through [braid_AccessStatusGet](#)*(..) routines.

The frequency of XBraid's calls to *access* is controlled through [braid_SetAccessLevel](#). For instance, if *access_level* is set to 2, then *access* is called every XBraid iteration and on every XBraid level. In this case, querying *status* to determine the current XBraid level and iteration will be useful. This scenario allows for even more detailed tracking of the simulation.

Eventually, access will be broadened to allow the user to steer XBraid.

12.1.2.3 typedef [braid_Int](#)(* [braid_PtFcnBufPack](#))(braid_App app, [braid_Vector](#) u, void *buffer, [braid_Int](#) *size_ptr)

This allows XBraid to send messages containing [braid_Vectors](#). This routine packs a vector u into a `void * buffer` for MPI.

12.1.2.4 `typedef braid_Int(* braid_PtFcnBufSize)(braid_App app,braid_Int *size_ptr)`

This routine tells XBraid message sizes by computing an upper bound in bytes for an arbitrary `braid_Vector`. This size must be an upper bound for what `BufPack` and `BufUnPack` will assume.

12.1.2.5 `typedef braid_Int(* braid_PtFcnBufUnpack)(braid_App app,void *buffer,braid_Vector *u_ptr)`

This allows XBraid to receive messages containing `braid_Vectors`. This routine unpacks a `void * buffer` from MPI into a `braid_Vector`.

12.1.2.6 `typedef braid_Int(* braid_PtFcnClone)(braid_App app,braid_Vector u,braid_Vector *v_ptr)`

Clone u into v_ptr

12.1.2.7 `typedef braid_Int(* braid_PtFcnCoarsen)(braid_App app,braid_Vector fu,braid_Vector *cu_ptr,braid_CoarsenRefStatus status)`

Spatial coarsening (optional). Allows the user to coarsen when going from a fine time grid to a coarse time grid. This function is called on every vector at each level, thus you can coarsen the entire space time domain. The action of this function should match the [braid_PtFcnRefine](#) function.

The user should query the status structure at run time with `braid_CoarsenRefGet**()` calls in order to determine how to coarsen. For instance, status tells you what the current time value is, and what the time step sizes on the fine and coarse levels are.

12.1.2.8 `typedef braid_Int(* braid_PtFcnFree)(braid_App app,braid_Vector u)`

Free and deallocate u

12.1.2.9 `typedef braid_Int(* braid_PtFcnInit)(braid_App app,braid_Real t,braid_Vector *u_ptr)`

Initializes a vector u_ptr at time t

12.1.2.10 `typedef braid_Int(* braid_PtFcnPhi)(braid_App app,braid_Vector u,braid_PhiStatus status)`

Defines the central time stepping function that the user must write. The user must advance the vector u from time $tstart$ to time $tstop$.

Query the status structure with `braid_PhiStatusGetTstart(status, &tstart)` and `braid_PhiStatusGetTstop(status, &tstop)` to get $tstart$ and $tstop$. The status structure also allows for steering. For example, `braid_PhiStatusSetRFactor(...)` allows for setting `rfactor`, which tells XBraid to refine this time interval.

12.1.2.11 `typedef braid_Int(* braid_PtFcnRefine)(braid_App app,braid_Vector cu,braid_Vector *fu_ptr,braid_CoarsenRefStatus status)`

Spatial refinement (optional). Allows the user to refine when going from a coarse time grid to a fine time grid. This function is called on every vector at each level, thus you can refine the entire space time domain. The action of this function should match the [braid_PtFcnCoarsen](#) function.

The user should query the status structure at run time with `braid_CoarsenRefGet**()` calls in order to determine how to coarsen. For instance, status tells you what the current time value is, and what the time step sizes on the fine and coarse levels are.

12.1.2.12 `typedef braid_Int(* braid_PtFcnSpatialNorm)(braid_App app,braid_Vector u,braid_Real *norm_ptr)`

Carry out a spatial norm by taking the norm of a `braid_Vector` $norm_ptr = || u ||$. A common choice is the standard Euclidean norm, but many other choices are possible, such as an L2-norm based on a finite element space. See [braid_SetTemporalNorm](#) for information on how the spatial norm is combined over time for a global space-time residual norm.

This global norm then controls halting.

12.1.2.13 `typedef braid_Int(* braid_PtFcnSum)(braid_App app,braid_Real alpha,braid_Vector x,braid_Real beta,braid_Vector y)`

AXPY, $\alpha x + \beta y \rightarrow y$

12.1.2.14 `typedef struct _braid_Vector_struct* braid_Vector`

This defines (roughly) a state vector at a certain time value. It could also contain any other information related to this vector which is needed to evolve the vector to the next time value, like mesh information.

12.2 User interface routines

Modules

- [General Interface routines](#)
- [XBraid status routines](#)

12.2.1 Detailed Description

These are all the user interface routines.

12.3 General Interface routines

Typedefs

- typedef struct _braid_Core_struct * [braid_Core](#)

Functions

- [braid_Int braid_Init](#) (MPI_Comm comm_world, MPI_Comm comm, [braid_Real](#) tstart, [braid_Real](#) tstop, [braid_Int](#) ntime, [braid_App](#) app, [braid_PtFcnPhi](#) phi, [braid_PtFcnInit](#) init, [braid_PtFcnClone](#) clone, [braid_PtFcnFree](#) free, [braid_PtFcnSum](#) sum, [braid_PtFcnSpatialNorm](#) spatialnorm, [braid_PtFcnAccess](#) access, [braid_PtFcnBufSize](#) bufsize, [braid_PtFcnBufPack](#) bufpack, [braid_PtFcnBufUnpack](#) bufunpack, [braid_Core](#) *core_ptr)
- [braid_Int braid_Drive](#) ([braid_Core](#) core)
- [braid_Int braid_Destroy](#) ([braid_Core](#) core)
- [braid_Int braid_PrintStats](#) ([braid_Core](#) core)
- [braid_Int braid_SetLoosexTol](#) ([braid_Core](#) core, [braid_Int](#) level, [braid_Real](#) loose_tol)
- [braid_Int braid_SetTightxTol](#) ([braid_Core](#) core, [braid_Int](#) level, [braid_Real](#) tight_tol)
- [braid_Int braid_SetMaxLevels](#) ([braid_Core](#) core, [braid_Int](#) max_levels)
- [braid_Int braid_SetMinCoarse](#) ([braid_Core](#) core, [braid_Int](#) min_coarse)
- [braid_Int braid_SetAbsTol](#) ([braid_Core](#) core, [braid_Real](#) atol)
- [braid_Int braid_SetRelTol](#) ([braid_Core](#) core, [braid_Real](#) rtol)
- [braid_Int braid_SetNRelax](#) ([braid_Core](#) core, [braid_Int](#) level, [braid_Int](#) nrelax)
- [braid_Int braid_SetCFactor](#) ([braid_Core](#) core, [braid_Int](#) level, [braid_Int](#) cfactor)
- [braid_Int braid_SetMaxIter](#) ([braid_Core](#) core, [braid_Int](#) max_iter)
- [braid_Int braid_SetFMG](#) ([braid_Core](#) core)
- [braid_Int braid_SetTemporalNorm](#) ([braid_Core](#) core, [braid_Int](#) tnorm)
- [braid_Int braid_SetNFMGVcyc](#) ([braid_Core](#) core, [braid_Int](#) nfmvg_Vcyc)
- [braid_Int braid_SetSpatialCoarsen](#) ([braid_Core](#) core, [braid_PtFcnCoarsen](#) coarsen)
- [braid_Int braid_SetSpatialRefine](#) ([braid_Core](#) core, [braid_PtFcnRefine](#) refine)
- [braid_Int braid_SetPrintLevel](#) ([braid_Core](#) core, [braid_Int](#) print_level)
- [braid_Int braid_SetPrintFile](#) ([braid_Core](#) core, const char *printfile_name)
- [braid_Int braid_SetAccessLevel](#) ([braid_Core](#) core, [braid_Int](#) access_level)
- [braid_Int braid_SplitCommworld](#) (const MPI_Comm *comm_world, [braid_Int](#) px, MPI_Comm *comm_x, MPI_Comm *comm_t)
- [braid_Int braid_GetNumIter](#) ([braid_Core](#) core, [braid_Int](#) *niter_ptr)
- [braid_Int braid_GetRNorm](#) ([braid_Core](#) core, [braid_Real](#) *rnorm_ptr)

12.3.1 Detailed Description

These are general interface routines, e.g., routines to initialize and run a XBraid solver, or to split a communicator into spatial and temporal components.

12.3.2 Typedef Documentation

12.3.2.1 typedef struct _braid_Core_struct* [braid_Core](#)

points to the core structure defined in [_braid.h](#)

12.3.3 Function Documentation

12.3.3.1 `braid_Int braid_Destroy (braid_Core core)`

Clean up and destroy core.

Parameters

<i>core</i>	braid_Core (_braid_Core) struct
-------------	---

12.3.3.2 **braid_Int braid_Drive (braid_Core core)**

Carry out a simulation with XBraid. Integrate in time.

Parameters

<i>core</i>	braid_Core (_braid_Core) struct
-------------	---

12.3.3.3 **braid_Int braid_GetNumIter (braid_Core core, braid_Int * niter_ptr)**

After Drive() finishes, this returns the number of iterations taken.

Parameters

<i>core</i>	braid_Core (_braid_Core) struct
<i>niter_ptr</i>	output, holds number of iterations taken

12.3.3.4 **braid_Int braid_GetRNorm (braid_Core core, braid_Real * rnorm_ptr)**

After Drive() finishes, this returns the last measured residual norm.

Parameters

<i>core</i>	braid_Core (_braid_Core) struct
<i>rnorm_ptr</i>	output, holds final residual norm

12.3.3.5 braid_Int braid_Init (MPI_Comm comm_world, MPI_Comm comm, braid_Real tstart, braid_Real tstop, braid_Int ntime, braid_App app, braid_PtFcnPhi phi, braid_PtFcnInit init, braid_PtFcnClone clone, braid_PtFcnFree free, braid_PtFcnSum sum, braid_PtFcnSpatialNorm spatialnorm, braid_PtFcnAccess access, braid_PtFcnBufSize bufsize, braid_PtFcnBufPack bufpack, braid_PtFcnBufUnpack bufunpack, braid_Core * core_ptr)

Create a core object with the required initial data.

This core is used by XBraid for internal data structures. The output is *core_ptr* which points to the newly created braid_Core structure.

Parameters

<i>comm_world</i>	Global communicator for space and time
<i>comm</i>	Communicator for temporal dimension
<i>tstart</i>	start time
<i>tstop</i>	End time
<i>ntime</i>	Initial number of temporal grid values
<i>app</i>	User-defined _braid_App structure
<i>phi</i>	User time stepping routine to advance a braid_Vector forward one step
<i>init</i>	Initialize a braid_Vector on the finest temporal grid

<i>clone</i>	Clone a <code>braid_Vector</code>
<i>free</i>	Free a <code>braid_Vector</code>
<i>sum</i>	Compute vector sum of two <code>braid_Vectors</code>
<i>spatialnorm</i>	Compute norm of a <code>braid_Vector</code> , this is a norm only over space
<i>access</i>	Allows access to XBraid and current <code>braid_Vector</code>
<i>bufsize</i>	Computes size for MPI buffer for one <code>braid_Vector</code>
<i>bufpack</i>	Packs MPI buffer to contain one <code>braid_Vector</code>
<i>bufunpack</i>	Unpacks MPI buffer into a <code>braid_Vector</code>
<i>core_ptr</i>	Pointer to <code>braid_Core</code> (<code>_braid_Core</code>) struct

12.3.3.6 `braid_Int braid_PrintStats (braid_Core core)`

Print statistics after a XBraid run.

Parameters

<i>core</i>	<code>braid_Core</code> (<code>_braid_Core</code>) struct
-------------	---

12.3.3.7 `braid_Int braid_SetAbsTol (braid_Core core, braid_Real atol)`

Set absolute stopping tolerance.

Recommended option over relative tolerance

Parameters

<i>core</i>	<code>braid_Core</code> (<code>_braid_Core</code>) struct
<i>atol</i>	absolute stopping tolerance

12.3.3.8 `braid_Int braid_SetAccessLevel (braid_Core core, braid_Int access_level)`

Set access level for XBraid. This controls how often the user's access routine is called.

- Level 0: Never call the user's access routine
- Level 1: Only call the user's access routine after XBraid is finished
- Level 2: Call the user's access routine every iteration and on every level. This is during `_braid_FRestrict`, during the down-cycle part of a XBraid iteration.

Default is level 1.

Parameters

<i>core</i>	<code>braid_Core</code> (<code>_braid_Core</code>) struct
<i>access_level</i>	desired <code>access_level</code>

12.3.3.9 `braid_Int braid_SetCFactor (braid_Core core, braid_Int level, braid_Int cfactor)`

Set the coarsening factor `cfactor` on grid `level` (level 0 is the finest grid). The default factor is 2 on all levels. To change the default factor, use `level = -1`.

Parameters

<i>core</i>	braid_Core (_braid_Core) struct
<i>level</i>	<i>level</i> to set coarsening factor on
<i>cfactor</i>	desired coarsening factor

12.3.3.10 **braid_Int** braid_SetFMG (**braid_Core** *core*)

Once called, XBraid will use FMG (i.e., F-cycles).

Parameters

<i>core</i>	braid_Core (_braid_Core) struct
-------------	---

12.3.3.11 **braid_Int** braid_SetLoosexTol (**braid_Core** *core*, **braid_Int** *level*, **braid_Real** *loose_tol*)

Set loose stopping tolerance *loose_tol* for spatial solves on grid *level* (level 0 is the finest grid).

Parameters

<i>core</i>	braid_Core (_braid_Core) struct
<i>level</i>	level to set <i>loose_tol</i>
<i>loose_tol</i>	tolerance to set

12.3.3.12 **braid_Int** braid_SetMaxIter (**braid_Core** *core*, **braid_Int** *max_iter*)

Set max number of multigrid iterations.

Parameters

<i>core</i>	braid_Core (_braid_Core) struct
<i>max_iter</i>	maximum iterations to allow

12.3.3.13 **braid_Int** braid_SetMaxLevels (**braid_Core** *core*, **braid_Int** *max_levels*)

Set max number of multigrid levels.

Parameters

<i>core</i>	braid_Core (_braid_Core) struct
<i>max_levels</i>	maximum levels to allow

12.3.3.14 **braid_Int** braid_SetMinCoarse (**braid_Core** *core*, **braid_Int** *min_coarse*)

Set minimum allowed coarse grid size. XBraid stops coarsening whenever creating the next coarser grid will result in a grid smaller than *min_coarse*. The maximum possible coarse grid size will be *min_coarse***coarsening_factor*.

Parameters

<i>core</i>	braid_Core (_braid_Core) struct
<i>min_coarse</i>	minimum coarse grid size

12.3.3.15 **braid_Int** braid_SetNFMGVcyc (**braid_Core** *core*, **braid_Int** *nfmvg_Vcyc*)

Set number of V-cycles to use at each FMG level (standard is 1)

Parameters

<i>core</i>	braid_Core (_braid_Core) struct
<i>nfmfg_Vcyc</i>	number of V-cycles to do each FMG level

12.3.3.16 **braid_Int braid_SetNRelax (braid_Core core, braid_Int level, braid_Int nrelax)**

Set the number of relaxation sweeps *nrelax* on grid *level* (level 0 is the finest grid). The default is 1 on all levels. To change the default factor, use *level* = -1. One sweep is a CF relaxation sweep.

Parameters

<i>core</i>	braid_Core (_braid_Core) struct
<i>level</i>	<i>level</i> to set <i>nrelax</i> on
<i>nrelax</i>	number of relaxations to do on <i>level</i>

12.3.3.17 **braid_Int braid_SetPrintFile (braid_Core core, const char * printfile_name)**

Set output file for runtime print messages. Level of printing is controlled by [braid_SetPrintLevel](#). Default is stdout.

Parameters

<i>core</i>	braid_Core (_braid_Core) struct
<i>printfile_name</i>	output file for XBraid runtime output

12.3.3.18 **braid_Int braid_SetPrintLevel (braid_Core core, braid_Int print_level)**

Set print level for XBraid. This controls how much information is printed to the XBraid print file ([braid_SetPrintFile](#)).

- Level 0: no output
- Level 1: print typical information like a residual history, number of levels in the XBraid hierarchy, and so on.
- Level 2: level 1 output, plus debug level output.

Default is level 1.

Parameters

<i>core</i>	braid_Core (_braid_Core) struct
<i>print_level</i>	desired print level

12.3.3.19 **braid_Int braid_SetRelTol (braid_Core core, braid_Real rtol)**

Set relative stopping tolerance, relative to the initial residual. Be careful. If your initial guess is all zero, then the initial residual may only be nonzero over one or two time values, and this will skew the relative tolerance. Absolute tolerances are recommended.

Parameters

<i>core</i>	braid_Core (_braid_Core) struct
<i>rtol</i>	relative stopping tolerance

12.3.3.20 **braid_Int braid_SetSpatialCoarsen (braid_Core core, braid_PtFcnCoarsen coarsen)**

Set spatial coarsening routine with user-defined routine. Default is no spatial refinement or coarsening.

Parameters

<i>core</i>	braid_Core (_braid_Core) struct
<i>coarsen</i>	function pointer to spatial coarsening routine

12.3.3.21 **braid_Int braid_SetSpatialRefine (braid_Core core, braid_PtFcnRefine refine)**

Set spatial refinement routine with user-defined routine. Default is no spatial refinement or coarsening.

Parameters

<i>core</i>	braid_Core (_braid_Core) struct
<i>refine</i>	function pointer to spatial refinement routine

12.3.3.22 **braid_Int braid_SetTemporalNorm (braid_Core core, braid_Int tnorm)**

Sets XBraid temporal norm.

This option determines how to obtain a global space-time residual norm. That is, this decides how to combine the spatial norms returned by [braid_PtFcnSpatialNorm](#) at each time step to obtain a global norm over space and time. It is this global norm that then controls halting.

There are three options for setting *tnorm*. See section [Halting tolerance](#) for a more detailed discussion (in [Introduction.md](#)).

- *tnorm=1*: One-norm summation of spatial norms
- *tnorm=2*: Two-norm summation of spatial norms
- *tnorm=3*: Infinity-norm combination of spatial norms

The default choice is *tnorm=2*

Parameters

<i>core</i>	braid_Core (_braid_Core) struct
<i>tnorm</i>	choice of temporal norm

12.3.3.23 **braid_Int braid_SetTightxTol (braid_Core core, braid_Int level, braid_Real tight_tol)**

Set tight stopping tolerance *tight_tol* for spatial solves on grid *level* (level 0 is the finest grid).

Parameters

<i>core</i>	braid_Core (_braid_Core) struct
<i>level</i>	level to set <i>tight_tol</i>
<i>tight_tol</i>	tolerance to set

12.3.3.24 **braid_Int braid_SplitCommworld (const MPI_Comm * comm_world, braid_Int px, MPI_Comm * comm_x, MPI_Comm * comm_t)**

Split MPI commworld into *comm_x* and *comm_t*, the spatial and temporal communicators. The total number of processors will equal $P_x \times P_t$, where P_x is the number of procs in space, and P_t is the number of procs in time.

Parameters

<i>comm_world</i>	Global communicator to split
<i>px</i>	Number of processors parallelizing space for a single time step
<i>comm_x</i>	Spatial communicator (written as output)
<i>comm_t</i>	Temporal communicator (written as output)

12.4 XBraid status routines

Data Structures

- struct [_braid_AccessStatus](#)
- struct [_braid_CoarsenRefStatus](#)
- struct [_braid_PhiStatus](#)

Macros

- `#define _braid_StatusElt(status, elt) ((status) -> elt)`

Typedefs

- typedef struct
 [_braid_AccessStatus_struct](#) * [braid_AccessStatus](#)
- typedef struct
 [_braid_CoarsenRefStatus_struct](#) * [braid_CoarsenRefStatus](#)
- typedef struct
 [_braid_PhiStatus_struct](#) * [braid_PhiStatus](#)

Functions

- [braid_Int _braid_AccessStatusInit](#) ([braid_Real](#) t, [braid_Real](#) rnorm, [braid_Int](#) iter, [braid_Int](#) level, [braid_Int](#) done, [braid_Int](#) wrapper_test, [braid_AccessStatus](#) status)
- [braid_Int _braid_AccessStatusDestroy](#) ([braid_AccessStatus](#) status)
- [braid_Int braid_AccessStatusGetT](#) ([braid_AccessStatus](#) status, [braid_Real](#) *t_ptr)
- [braid_Int braid_AccessStatusGetResidual](#) ([braid_AccessStatus](#) status, [braid_Real](#) *rnorm_ptr)
- [braid_Int braid_AccessStatusGetIter](#) ([braid_AccessStatus](#) status, [braid_Int](#) *iter_ptr)
- [braid_Int braid_AccessStatusGetLevel](#) ([braid_AccessStatus](#) status, [braid_Int](#) *level_ptr)
- [braid_Int braid_AccessStatusGetDone](#) ([braid_AccessStatus](#) status, [braid_Int](#) *done_ptr)
- [braid_Int braid_AccessStatusGetWrapperTest](#) ([braid_AccessStatus](#) status, [braid_Int](#) *wtest_ptr)
- [braid_Int braid_AccessStatusGetTILD](#) ([braid_AccessStatus](#) status, [braid_Real](#) *t_ptr, [braid_Int](#) *iter_ptr, [braid_Int](#) *level_ptr, [braid_Int](#) *done_ptr)
- [braid_Int _braid_CoarsenRefStatusInit](#) ([braid_Real](#) tstart, [braid_Real](#) f_tprior, [braid_Real](#) f_tstop, [braid_Real](#) c_tprior, [braid_Real](#) c_tstop, [braid_CoarsenRefStatus](#) status)
- [braid_Int _braid_CoarsenRefStatusDestroy](#) ([braid_CoarsenRefStatus](#) status)
- [braid_Int braid_CoarsenRefStatusGetTstart](#) ([braid_CoarsenRefStatus](#) status, [braid_Real](#) *tstart_ptr)
- [braid_Int braid_CoarsenRefStatusGetFTstop](#) ([braid_CoarsenRefStatus](#) status, [braid_Real](#) *f_tstop_ptr)
- [braid_Int braid_CoarsenRefStatusGetFTprior](#) ([braid_CoarsenRefStatus](#) status, [braid_Real](#) *f_tprior_ptr)
- [braid_Int braid_CoarsenRefStatusGetCTstop](#) ([braid_CoarsenRefStatus](#) status, [braid_Real](#) *c_tstop_ptr)
- [braid_Int braid_CoarsenRefStatusGetCTprior](#) ([braid_CoarsenRefStatus](#) status, [braid_Real](#) *c_tprior_ptr)
- [braid_Int braid_CoarsenRefStatusGetTpriorTstop](#) ([braid_CoarsenRefStatus](#) status, [braid_Real](#) *tstart_ptr, [braid_Real](#) *f_tprior_ptr, [braid_Real](#) *f_tstop_ptr, [braid_Real](#) *c_tprior_ptr, [braid_Real](#) *c_tstop_ptr)
- [braid_Int _braid_PhiStatusInit](#) ([braid_Real](#) tstart, [braid_Real](#) tstop, [braid_Real](#) accuracy, [braid_PhiStatus](#) status)
- [braid_Int _braid_PhiStatusDestroy](#) ([braid_PhiStatus](#) status)
- [braid_Int braid_PhiStatusGetTstart](#) ([braid_PhiStatus](#) status, [braid_Real](#) *tstart_ptr)
- [braid_Int braid_PhiStatusGetTstop](#) ([braid_PhiStatus](#) status, [braid_Real](#) *tstop_ptr)
- [braid_Int braid_PhiStatusGetAccuracy](#) ([braid_PhiStatus](#) status, [braid_Real](#) *accuracy_ptr)
- [braid_Int braid_PhiStatusSetRFactor](#) ([braid_PhiStatus](#) status, [braid_Real](#) rfactor)
- [braid_Int braid_PhiStatusGetTstartTstop](#) ([braid_PhiStatus](#) status, [braid_Real](#) *tstart_ptr, [braid_Real](#) *tstop_ptr)

12.4.1 Detailed Description

XBraid status structures are what tell the user the status of the simulation when their routines (phi, coarsen/refine, access) are called.

12.4.2 Macro Definition Documentation

12.4.2.1 `#define _braid_StatusElt(status, elt) ((status)-> elt)`

Accessor for all *braid***Status attributes

12.4.3 Typedef Documentation

12.4.3.1 `typedef struct _braid_AccessStatus_struct* braid_AccessStatus`

The user access routine will receive *braid_AccessStatus*, which will be a pointer to the actual *_braid_AccessStatus_struct*

12.4.3.2 `typedef struct _braid_CoarsenRefStatus_struct* braid_CoarsenRefStatus`

The user coarsen and refine routines will receive *braid_CoarsenRefStatus*, which will be a pointer to the actual *_braid_CoarsenRefStatus_struct*

12.4.3.3 `typedef struct _braid_PhiStatus_struct* braid_PhiStatus`

The user's phi routine will receive *braid_PhiStatus*, which will be a pointer to the actual *_braid_PhiStatus_struct*

12.4.4 Function Documentation

12.4.4.1 `braid_Int _braid_AccessStatusDestroy (braid_AccessStatus status)`

Destroy a *braid_AccessStatus* structurestructure to be destroyed

12.4.4.2 `braid_Int _braid_AccessStatusInit (braid_Real t, braid_Real rnorm, braid_Int iter, braid_Int level, braid_Int done, braid_Int wrapper_test, braid_AccessStatus status)`

Initialize a *braid_AccessStatus* structure

Parameters

<i>t</i>	current time
<i>rnorm</i>	current residual norm in XBraid
<i>iter</i>	current iteration in XBraid
<i>level</i>	current level in XBraid
<i>done</i>	boolean describing whether XBraid has finished
<i>wrapper_test</i>	boolean describing whether this call is only a wrapper test
<i>status</i>	structure to initialize

12.4.4.3 `braid_Int _braid_CoarsenRefStatusDestroy (braid_CoarsenRefStatus status)`

Destroy a *braid_CoarsenRefStatus* structurestructure to be destroyed

12.4.4.4 **braid_Int _braid_CoarsenRefStatusInit (braid_Real *tstart*, braid_Real *f_tprior*, braid_Real *f_tstop*, braid_Real *c_tprior*, braid_Real *c_tstop*, braid_CoarsenRefStatus *status*)**

Initialize a braid_CoarsenRefStatus structure

Parameters

<i>tstart</i>	time value for current vector
<i>f_tprior</i>	time value to the left of tstart on fine grid
<i>f_tstop</i>	time value to the right of tstart on fine grid
<i>c_tprior</i>	time value to the left of tstart on coarse grid
<i>c_tstop</i>	time value to the right of tstart on coarse grid
<i>status</i>	structure to initialize

12.4.4.5 **braid_Int _braid_PhiStatusDestroy (braid_PhiStatus status)**

Destroy a braid_PhiStatus structurestructure to be destroyed

12.4.4.6 **braid_Int _braid_PhiStatusInit (braid_Real tstart, braid_Real tstop, braid_Real accuracy, braid_PhiStatus status)**

Initialize a braid_PhiStatus structure

Parameters

<i>tstart</i>	current time value
<i>tstop</i>	time value to evolve towards, time value to the right of tstart
<i>accuracy</i>	advanced option allowing variable accuracy for implicit phi
<i>status</i>	structure to initialize

12.4.4.7 **braid_Int braid_AccessStatusGetDone (braid_AccessStatus status, braid_Int * done_ptr)**

Return whether XBraid is done for the current simulation.

done_ptr = 1 indicates that XBraid has finished iterating, (either maxiter has been reached, or the tolerance has been met).

Parameters

<i>status</i>	structure containing current simulation info
<i>done_ptr</i>	output, =1 if XBraid has finished, else =0

12.4.4.8 **braid_Int braid_AccessStatusGetIter (braid_AccessStatus status, braid_Int * iter_ptr)**

Return the current iteration from the AccessStatus structure.

Parameters

<i>status</i>	structure containing current simulation info
<i>iter_ptr</i>	output, current XBraid iteration number

12.4.4.9 **braid_Int braid_AccessStatusGetLevel (braid_AccessStatus status, braid_Int * level_ptr)**

Return the current XBraid level from the AccessStatus structure.

Parameters

<i>status</i>	structure containing current simulation info
---------------	--

<i>level_ptr</i>	output, current level in XBraid
------------------	---------------------------------

12.4.4.10 **braid_Int** braid_AccessStatusGetResidual (**braid_AccessStatus** *status*, **braid_Real** * *rnorm_ptr*)

Return the current residual norm from the AccessStatus structure.

Parameters

<i>status</i>	structure containing current simulation info
<i>rnorm_ptr</i>	output, current residual norm

12.4.4.11 **braid_Int** braid_AccessStatusGetT (**braid_AccessStatus** *status*, **braid_Real** * *t_ptr*)

Return the current time from the AccessStatus structure.

Parameters

<i>status</i>	structure containing current simulation info
<i>t_ptr</i>	output, current time

12.4.4.12 **braid_Int** braid_AccessStatusGetTILD (**braid_AccessStatus** *status*, **braid_Real** * *t_ptr*, **braid_Int** * *iter_ptr*, **braid_Int** * *level_ptr*, **braid_Int** * *done_ptr*)

Return XBraid status for the current simulation. Four values are returned.

TILD : time, iteration, level, done

These values are also available through individual Get routines. These individual routines are the location of detailed documentation on each parameter, e.g., see *braid_AccessStatusGetDone* for more information on the *done* value.

Parameters

<i>status</i>	structure containing current simulation info
<i>t_ptr</i>	output, current time
<i>iter_ptr</i>	output, current iteration in XBraid
<i>level_ptr</i>	output, current level in XBraid
<i>done_ptr</i>	output, boolean describing whether XBraid has finished

12.4.4.13 **braid_Int** braid_AccessStatusGetWrapperTest (**braid_AccessStatus** *status*, **braid_Int** * *wtest_ptr*)

Return whether this is a wrapper test or an XBraid run

Parameters

<i>status</i>	structure containing current simulation info
<i>wtest_ptr</i>	output, =1 if this is a wrapper test, =0 if XBraid run

12.4.4.14 **braid_Int** braid_CoarsenRefStatusGetCTprior (**braid_CoarsenRefStatus** *status*, **braid_Real** * *c_tprior_ptr*)

Return the **coarse grid** time value to the left of the current time value from the CoarsenRefStatus structure.

Parameters

<i>status</i>	structure containing current simulation info
<i>c_tprior_ptr</i>	output, time value to the left of current time value on coarse grid

12.4.4.15 **braid_Int** **braid_CoarsenRefStatusGetCTstop** (**braid_CoarsenRefStatus** *status*, **braid_Real** * *c_tstop_ptr*)

Return the **coarse grid** time value to the right of the current time value from the CoarsenRefStatus structure.

Parameters

<i>status</i>	structure containing current simulation info
<i>c_tstop_ptr</i>	output, time value to the right of current time value on coarse grid

12.4.4.16 **braid_Int** **braid_CoarsenRefStatusGetFTprior** (**braid_CoarsenRefStatus** *status*, **braid_Real** * *f_tprior_ptr*)

Return the **fine grid** time value to the left of the current time value from the CoarsenRefStatus structure.

Parameters

<i>status</i>	structure containing current simulation info
<i>f_tprior_ptr</i>	output, time value to the left of current time value on fine grid

12.4.4.17 **braid_Int** **braid_CoarsenRefStatusGetFTstop** (**braid_CoarsenRefStatus** *status*, **braid_Real** * *f_tstop_ptr*)

Return the **fine grid** time value to the right of the current time value from the CoarsenRefStatus structure.

Parameters

<i>status</i>	structure containing current simulation info
<i>f_tstop_ptr</i>	output, time value to the right of current time value on fine grid

12.4.4.18 **braid_Int** **braid_CoarsenRefStatusGetTpriorTstop** (**braid_CoarsenRefStatus** *status*, **braid_Real** * *tstart_ptr*, **braid_Real** * *f_tprior_ptr*, **braid_Real** * *f_tstop_ptr*, **braid_Real** * *c_tprior_ptr*, **braid_Real** * *c_tstop_ptr*)

Return XBraid status for the current simulation. Five values are returned, *tstart*, *f_tprior*, *f_tstop*, *c_tprior*, *c_tstop*.

These values are also available through individual Get routines. These individual routines are the location of detailed documentation on each parameter, e.g., see *braid_CoarsenRefStatusGetCTprior* for more information on the *c_tprior* value.

Parameters

<i>status</i>	structure containing current simulation info
<i>tstart_ptr</i>	output, time value current vector
<i>f_tprior_ptr</i>	output, time value to the left of <i>tstart</i> on fine grid
<i>f_tstop_ptr</i>	output, time value to the right of <i>tstart</i> on fine grid
<i>c_tprior_ptr</i>	output, time value to the left of <i>tstart</i> on coarse grid
<i>c_tstop_ptr</i>	output, time value to the right of <i>tstart</i> on coarse grid

12.4.4.19 **braid_Int** **braid_CoarsenRefStatusGetTstart** (**braid_CoarsenRefStatus** *status*, **braid_Real** * *tstart_ptr*)

Return the current time value from the CoarsenRefStatus structure.

Parameters

<i>status</i>	structure containing current simulation info
<i>tstart_ptr</i>	output, current time

12.4.4.20 **braid_Int braid_PhiStatusGetAccuracy (braid_PhiStatus *status*, braid_Real * *accuracy_ptr*)**

Return the current accuracy value, usually between 0 and 1.0, which can allow for tuning of implicit solve accuracy

Parameters

<i>status</i>	structure containing current simulation info
<i>accuracy_ptr</i>	output, current accuracy value

12.4.4.21 **braid_Int braid_PhiStatusGetTstart (braid_PhiStatus *status*, braid_Real * *tstart_ptr*)**

Return the current time value from the PhiStatus structure.

Parameters

<i>status</i>	structure containing current simulation info
<i>tstart_ptr</i>	output, current time

12.4.4.22 **braid_Int braid_PhiStatusGetTstartTstop (braid_PhiStatus *status*, braid_Real * *tstart_ptr*, braid_Real * *tstop_ptr*)**

Return XBraid status for the current simulation. Two values are returned, *tstart* and *tstop*.

These values are also available through individual Get routines. These individual routines are the location of detailed documentation on each parameter, e.g., see *braid_PhiStatusGetTstart* for more information on the *tstart* value.

Parameters

<i>status</i>	structure containing current simulation info
<i>tstart_ptr</i>	output, current time
<i>tstop_ptr</i>	output, next time value to evolve towards

12.4.4.23 **braid_Int braid_PhiStatusGetTstop (braid_PhiStatus *status*, braid_Real * *tstop_ptr*)**

Return the time value to the right of the current time value from the PhiStatus structure.

Parameters

<i>status</i>	structure containing current simulation info
<i>tstop_ptr</i>	output, next time value to evolve towards

12.4.4.24 **braid_Int braid_PhiStatusSetRFactor (braid_PhiStatus *status*, braid_Real *rfactor*)**

Set the *rfactor*, a desired refinement factor for this interval. *rfactor*=1 indicates no refinement, otherwise, this interval is subdivided *rfactor* times.

Parameters

<i>status</i>	structure containing current simulation info
<i>rfactor</i>	user-determined desired rfactor

12.5 XBraid test routines

Functions

- `braid_Int braid_TestInitAccess` (`braid_App` app, `MPI_Comm` comm_x, `FILE *fp`, `braid_Real` t, `braid_PtFcnInit` init, `braid_PtFcnAccess` access, `braid_PtFcnFree` free)
- `braid_Int braid_TestClone` (`braid_App` app, `MPI_Comm` comm_x, `FILE *fp`, `braid_Real` t, `braid_PtFcnInit` init, `braid_PtFcnAccess` access, `braid_PtFcnFree` free, `braid_PtFcnClone` clone)
- `braid_Int braid_TestSum` (`braid_App` app, `MPI_Comm` comm_x, `FILE *fp`, `braid_Real` t, `braid_PtFcnInit` init, `braid_PtFcnAccess` access, `braid_PtFcnFree` free, `braid_PtFcnClone` clone, `braid_PtFcnSum` sum)
- `braid_Int braid_TestSpatialNorm` (`braid_App` app, `MPI_Comm` comm_x, `FILE *fp`, `braid_Real` t, `braid_PtFcnInit` init, `braid_PtFcnFree` free, `braid_PtFcnClone` clone, `braid_PtFcnSum` sum, `braid_PtFcnSpatialNorm` spatialnorm)
- `braid_Int braid_TestBuf` (`braid_App` app, `MPI_Comm` comm_x, `FILE *fp`, `braid_Real` t, `braid_PtFcnInit` init, `braid_PtFcnFree` free, `braid_PtFcnSum` sum, `braid_PtFcnSpatialNorm` spatialnorm, `braid_PtFcnBufSize` bufsize, `braid_PtFcnBufPack` bufpack, `braid_PtFcnBufUnpack` bufunpack)
- `braid_Int braid_TestCoarsenRefine` (`braid_App` app, `MPI_Comm` comm_x, `FILE *fp`, `braid_Real` t, `braid_Real` fdt, `braid_Real` cdt, `braid_PtFcnInit` init, `braid_PtFcnAccess` access, `braid_PtFcnFree` free, `braid_PtFcnClone` clone, `braid_PtFcnSum` sum, `braid_PtFcnSpatialNorm` spatialnorm, `braid_PtFcnCoarsen` coarsen, `braid_PtFcnRefine` refine)
- `braid_Int braid_TestAll` (`braid_App` app, `MPI_Comm` comm_x, `FILE *fp`, `braid_Real` t, `braid_Real` fdt, `braid_Real` cdt, `braid_PtFcnInit` init, `braid_PtFcnFree` free, `braid_PtFcnClone` clone, `braid_PtFcnSum` sum, `braid_PtFcnSpatialNorm` spatialnorm, `braid_PtFcnBufSize` bufsize, `braid_PtFcnBufPack` bufpack, `braid_PtFcnBufUnpack` bufunpack, `braid_PtFcnCoarsen` coarsen, `braid_PtFcnRefine` refine)

12.5.1 Detailed Description

These are sanity check routines to help a user test their XBraid code.

12.5.2 Function Documentation

12.5.2.1 `braid_Int braid_TestAll` (`braid_App` app, `MPI_Comm` comm_x, `FILE *fp`, `braid_Real` t, `braid_Real` fdt, `braid_Real` cdt, `braid_PtFcnInit` init, `braid_PtFcnFree` free, `braid_PtFcnClone` clone, `braid_PtFcnSum` sum, `braid_PtFcnSpatialNorm` spatialnorm, `braid_PtFcnBufSize` bufsize, `braid_PtFcnBufPack` bufpack, `braid_PtFcnBufUnpack` bufunpack, `braid_PtFcnCoarsen` coarsen, `braid_PtFcnRefine` refine)

Runs all of the individual `braid_Test*` routines

- Returns 0 if the tests fail
- Returns 1 if the tests pass
- Check the log messages to see details of which tests failed.

Parameters

<i>app</i>	User defined App structure
<i>comm_x</i>	Spatial communicator
<i>fp</i>	File pointer (could be stdout or stderr) for log messages

<i>t</i>	Time value to initialize test vectors with
<i>fdt</i>	Fine time step value that you spatially coarsen from
<i>cdt</i>	Coarse time step value that you coarsen to
<i>init</i>	Initialize a braid_Vector on finest temporal grid
<i>free</i>	Free a braid_Vector
<i>clone</i>	Clone a braid_Vector
<i>sum</i>	Compute vector sum of two braid_Vectors
<i>spatialnorm</i>	Compute norm of a braid_Vector, this is a norm only over space
<i>bufsize</i>	Computes size in bytes for one braid_Vector MPI buffer
<i>bufpack</i>	Packs MPI buffer to contain one braid_Vector
<i>bufunpack</i>	Unpacks MPI buffer into a braid_Vector
<i>coarsen</i>	Spatially coarsen a vector. If NULL, test is skipped.
<i>refine</i>	Spatially refine a vector. If NULL, test is skipped.

12.5.2.2 **braid_Int braid_TestBuf (braid_App app, MPI_Comm comm_x, FILE * fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnFree free, braid_PtFcnSum sum, braid_PtFcnSpatialNorm spatialnorm, braid_PtFcnBufSize bufsize, braid_PtFcnBufPack bufpack, braid_PtFcnBufUnpack bufunpack)**

Test the BufPack, BufUnpack and BufSize functions.

A vector is initialized at time *t*, packed into a buffer, then unpacked from a buffer. The unpacked result must equal the original vector.

- Returns 0 if the tests fail
- Returns 1 if the tests pass
- Check the log messages to see details of which tests failed.

Parameters

<i>app</i>	User defined App structure
<i>comm_x</i>	Spatial communicator
<i>fp</i>	File pointer (could be stdout or stderr) for log messages
<i>t</i>	Time value to test Buffer routines (used to initialize the vectors)
<i>init</i>	Initialize a braid_Vector on finest temporal grid
<i>free</i>	Free a braid_Vector
<i>sum</i>	Compute vector sum of two braid_Vectors
<i>spatialnorm</i>	Compute norm of a braid_Vector, this is a norm only over space
<i>bufsize</i>	Computes size in bytes for one braid_Vector MPI buffer
<i>bufpack</i>	Packs MPI buffer to contain one braid_Vector
<i>bufunpack</i>	Unpacks MPI buffer containing one braid_Vector

12.5.2.3 **braid_Int braid_TestClone (braid_App app, MPI_Comm comm_x, FILE * fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnAccess access, braid_PtFcnFree free, braid_PtFcnClone clone)**

Test the clone function.

A vector is initialized at time *t*, cloned, and both vectors are written. Then both vectors are free-d. The user is to check (via the access function) to see if it is identical.

Parameters

<i>app</i>	User defined App structure
<i>comm_x</i>	Spatial communicator
<i>fp</i>	File pointer (could be stdout or stderr) for log messages
<i>t</i>	Time value to test clone with (used to initialize the vectors)
<i>init</i>	Initialize a braid_Vector on finest temporal grid
<i>access</i>	Allows access to XBraid and current braid_Vector (can be NULL for no writing)
<i>free</i>	Free a braid_Vector
<i>clone</i>	Clone a braid_Vector

12.5.2.4 **braid_Int braid_TestCoarsenRefine (braid_App app, MPI_Comm comm_x, FILE * fp, braid_Real t, braid_Real fdt, braid_Real cdt, braid_PtFcnInit init, braid_PtFcnAccess access, braid_PtFcnFree free, braid_PtFcnClone clone, braid_PtFcnSum sum, braid_PtFcnSpatialNorm spatialnorm, braid_PtFcnCoarsen coarsen, braid_PtFcnRefine refine)**

Test the Coarsen and Refine functions.

A vector is initialized at time t , and various sanity checks on the spatial coarsening and refinement routines are run.

- Returns 0 if the tests fail
- Returns 1 if the tests pass
- Check the log messages to see details of which tests failed.

Parameters

<i>app</i>	User defined App structure
<i>comm_x</i>	Spatial communicator
<i>fp</i>	File pointer (could be stdout or stderr) for log messages
<i>t</i>	Time value to initialize test vectors
<i>fdt</i>	Fine time step value that you spatially coarsen from
<i>cdt</i>	Coarse time step value that you coarsen to
<i>init</i>	Initialize a braid_Vector on finest temporal grid
<i>access</i>	Allows access to XBraid and current braid_Vector (can be NULL for no writing)
<i>free</i>	Free a braid_Vector
<i>clone</i>	Clone a braid_Vector
<i>sum</i>	Compute vector sum of two braid_Vectors
<i>spatialnorm</i>	Compute norm of a braid_Vector, this is a norm only over space
<i>coarsen</i>	Spatially coarsen a vector
<i>refine</i>	Spatially refine a vector

12.5.2.5 **braid_Int braid_TestInitAccess (braid_App app, MPI_Comm comm_x, FILE * fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnAccess access, braid_PtFcnFree free)**

Test the init, access and free functions.

A vector is initialized at time t , written, and then free-d

Parameters

<i>app</i>	User defined App structure
<i>comm_x</i>	Spatial communicator
<i>fp</i>	File pointer (could be stdout or stderr) for log messages
<i>t</i>	Time value to test init with (used to initialize the vectors)
<i>init</i>	Initialize a braid_Vector on finest temporal grid
<i>access</i>	Allows access to XBraid and current braid_Vector (can be NULL for no writing)
<i>free</i>	Free a braid_Vector

12.5.2.6 `braid_Int braid_TestSpatialNorm (braid_App app, MPI_Comm comm_x, FILE * fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnFree free, braid_PtFcnClone clone, braid_PtFcnSum sum, braid_PtFcnSpatialNorm spatialnorm)`

Test the spatialnorm function.

A vector is initialized at time *t* and then cloned. Various norm evaluations like $\| 3 v \| / \| v \|$ with known output are then done.

- Returns 0 if the tests fail
- Returns 1 if the tests pass
- Check the log messages to see details of which tests failed.

Parameters

<i>app</i>	User defined App structure
<i>comm_x</i>	Spatial communicator
<i>fp</i>	File pointer (could be stdout or stderr) for log messages
<i>t</i>	Time value to test SpatialNorm with (used to initialize the vectors)
<i>init</i>	Initialize a braid_Vector on finest temporal grid
<i>free</i>	Free a braid_Vector
<i>clone</i>	Clone a braid_Vector
<i>sum</i>	Compute vector sum of two braid_Vectors
<i>spatialnorm</i>	Compute norm of a braid_Vector, this is a norm only over space

12.5.2.7 `braid_Int braid_TestSum (braid_App app, MPI_Comm comm_x, FILE * fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnAccess access, braid_PtFcnFree free, braid_PtFcnClone clone, braid_PtFcnSum sum)`

Test the sum function.

A vector is initialized at time *t*, cloned, and then these two vectors are summed a few times, with the results written. The vectors are then free-d. The user is to check (via the access function) that the output matches the sum of the two original vectors.

Parameters

<i>app</i>	User defined App structure
<i>comm_x</i>	Spatial communicator
<i>fp</i>	File pointer (could be stdout or stderr) for log messages
<i>t</i>	Time value to test Sum with (used to initialize the vectors)
<i>init</i>	Initialize a braid_Vector on finest temporal grid

<i>access</i>	Allows access to XBraid and current braid_Vector (can be NULL for no writing)
<i>free</i>	Free a braid_Vector
<i>clone</i>	Clone a braid_Vector
<i>sum</i>	Compute vector sum of two braid_Vectors

13 Data Structure Documentation

13.1 `_braid_AccessStatus` Struct Reference

Data Fields

- [braid_Real t](#)
- [braid_Int iter](#)
- [braid_Int level](#)
- [braid_Real rnorm](#)
- [braid_Int done](#)
- [braid_Int wrapper_test](#)

13.1.1 Detailed Description

`AccessStatus` structure which defines the status of XBraid at a given instant on some level during a run. The user accesses it through `braid_AccessStatusGet**()` functions.

13.1.2 Field Documentation

13.1.2.1 `braid_Int done`

boolean describing whether XBraid has finished

13.1.2.2 `braid_Int iter`

XBraid iteration number

13.1.2.3 `braid_Int level`

current level in XBraid

13.1.2.4 `braid_Real rnorm`

residual norm

13.1.2.5 `braid_Real t`

current time

13.1.2.6 `braid_Int wrapper_test`

boolean describing whether this call is only a wrapper test

The documentation for this struct was generated from the following file:

- [braid_status.h](#)

13.2 `_braid_AccuracyHandle` Struct Reference

Data Fields

- [braid_Int matchF](#)

- [braid_Real value](#)
- [braid_Real old_value](#)
- [braid_Real loose](#)
- [braid_Real tight](#)
- [braid_Int tight_used](#)

13.2.1 Detailed Description

XBraid Accuracy Handle, used for controlling the accuracy of solves during implicit time stepping. For instance, to do less accurate solves on coarse time grids

13.2.2 Field Documentation

13.2.2.1 [braid_Real loose](#)

loose accuracy for spatial solves

13.2.2.2 [braid_Int matchF](#)

13.2.2.3 [braid_Real old_value](#)

old accuracy value used in FRestrict

13.2.2.4 [braid_Real tight](#)

tight accuracy for spatial solves

13.2.2.5 [braid_Int tight_used](#)

tight accuracy used (1) or not (0)

13.2.2.6 [braid_Real value](#)

accuracy value

The documentation for this struct was generated from the following file:

- [_braid.h](#)

13.3 [_braid_CoarsenRefStatus](#) Struct Reference

Data Fields

- [braid_Real tstart](#)
- [braid_Real f_tprior](#)
- [braid_Real f_tstop](#)
- [braid_Real c_tprior](#)
- [braid_Real c_tstop](#)

13.3.1 Detailed Description

The user coarsen and refine routines will receive a `CoarsenRefStatus` structure, which defines the status of XBraid at a given instant of coarsening or refinement on some level during a run. The user accesses it through `braid_CoarsenRefStatusGet**()` functions.

13.3.2 Field Documentation

13.3.2.1 `braid_Real c_tprior`

time value to the left of tstart on coarse grid

13.3.2.2 `braid_Real c_tstop`

time value to the right of tstart on coarse grid

13.3.2.3 `braid_Real f_tprior`

time value to the left of tstart on fine grid

13.3.2.4 `braid_Real f_tstop`

time value to the right of tstart on fine grid

13.3.2.5 `braid_Real tstart`

current time value

The documentation for this struct was generated from the following file:

- [braid_status.h](#)

13.4 `_braid_CommHandle` Struct Reference

Data Fields

- [braid_Int request_type](#)
- [braid_Int num_requests](#)
- `MPI_Request *` [requests](#)
- `MPI_Status *` [status](#)
- `void *` [buffer](#)
- [braid_Vector *](#) [vector_ptr](#)

13.4.1 Detailed Description

XBraid comm handle structure

Used for initiating and completing nonblocking communication to pass `braid_Vectors` between processors.

13.4.2 Field Documentation

13.4.2.1 void* buffer

Buffer for message

13.4.2.2 braid_Int num_requests

number of active requests for this handle, usually 1

13.4.2.3 braid_Int request_type

two values: recv type = 1, and send type = 0

13.4.2.4 MPI_Request* requests

MPI request structure

13.4.2.5 MPI_Status* status

MPI status

13.4.2.6 braid_Vector* vector_ptr

braid_vector being sent/received

The documentation for this struct was generated from the following file:

- [_braid.h](#)

13.5 _braid_Core Struct Reference

Data Fields

- MPI_Comm [comm_world](#)
- MPI_Comm [comm](#)
- [braid_Real](#) [tstart](#)
- [braid_Real](#) [tstop](#)
- [braid_Int](#) [ntime](#)
- [braid_App](#) [app](#)
- [braid_PtFcnPhi](#) [phi](#)
- [braid_PtFcnInit](#) [init](#)
- [braid_PtFcnClone](#) [clone](#)
- [braid_PtFcnFree](#) [free](#)
- [braid_PtFcnSum](#) [sum](#)
- [braid_PtFcnSpatialNorm](#) [spatialnorm](#)
- [braid_PtFcnAccess](#) [access](#)
- [braid_PtFcnBufSize](#) [bufsize](#)
- [braid_PtFcnBufPack](#) [bufpack](#)
- [braid_PtFcnBufUnpack](#) [bufunpack](#)
- [braid_PtFcnCoarsen](#) [coarsen](#)
- [braid_PtFcnRefine](#) [refine](#)
- [braid_Int](#) [access_level](#)
- [braid_Int](#) [print_level](#)
- [braid_Int](#) [max_levels](#)

- `braid_Int` min_coarse
- `braid_Real` tol
- `braid_Int` rtol
- `braid_Int` * nrels
- `braid_Int` nrdefault
- `braid_Int` * cfactors
- `braid_Int` cfdefault
- `braid_Int` max_iter
- `braid_Int` niter
- `braid_Real` rnorm
- `braid_Int` fmg
- `braid_Int` nfmg_Vcyc
- `braid_Int` tnorm
- `braid_Real` * tnorm_a
- `_braid_AccuracyHandle` * accuracy
- `braid_AccessStatus` astatus
- `braid_CoarsenRefStatus` cstatus
- `braid_PhiStatus` pstatus
- `braid_Int` gupper
- `braid_Int` * rfactors
- `braid_Int` nlevels
- `_braid_Grid` ** grids
- `braid_Real` localtime
- `braid_Real` globaltime

13.5.1 Detailed Description

The typedef `_braid_Core` struct is a **critical** part of XBraid and is passed to *each* routine in XBraid. It thus allows each routine access to XBraid attributes.

13.5.2 Field Documentation

13.5.2.1 `braid_PtFcnAccess` access

user access function to XBraid and current vector

13.5.2.2 `braid_Int` access_level

determines how often to call the user's access routine

13.5.2.3 `_braid_AccuracyHandle`* accuracy

accuracy of spatial solves on different levels

13.5.2.4 `braid_App` app

application data for the user

13.5.2.5 `braid_AccessStatus` astatus

status structure passed to user-written Access routine

13.5.2.6 braid_PtFcnBufPack bufpack

pack a buffer

13.5.2.7 braid_PtFcnBufSize bufsize

return buffer size

13.5.2.8 braid_PtFcnBufUnpack bufunpack

unpack a buffer

13.5.2.9 braid_Int* cfactors

coarsening factors

13.5.2.10 braid_Int cfdefault

default coarsening factor

13.5.2.11 braid_PtFcnClone clone

clone a vector

13.5.2.12 braid_PtFcnCoarsen coarsen

(optional) return a coarsened vector

13.5.2.13 MPI_Comm comm

communicator for the time dimension

13.5.2.14 MPI_Comm comm_world**13.5.2.15 braid_CoarsenRefStatus cstatus**

status structure passed to user-written coarsen/refine routines

13.5.2.16 braid_Int fmg

use FMG cycle

13.5.2.17 braid_PtFcnFree free

free up a vector

13.5.2.18 braid_Real globaltime

global wall time for [braid_Drive\(\)](#)

13.5.2.19 _braid_Grid grids**

pointer to temporal grid structures for each level

13.5.2.20 braid_Int gupper

global upper index on the fine grid

13.5.2.21 braid_PtFcnInit init

return an initialized `braid_Vector`

13.5.2.22 braid_Real localtime

local wall time for `braid_Drive()`

13.5.2.23 braid_Int max_iter

maximum number of multigrid in time iterations

13.5.2.24 braid_Int max_levels

maximum number of temporal grid levels

13.5.2.25 braid_Int min_coarse

minimum possible coarse grid size

13.5.2.26 braid_Int nfmfg_Vcyc

number of V-cycle calls at each level in FMG

13.5.2.27 braid_Int niter

number of iterations

13.5.2.28 braid_Int nlevels

number of temporal grid levels

13.5.2.29 braid_Int nrdefault

default number of pre-relaxations

13.5.2.30 braid_Int* nrels

number of pre-relaxations on each level

13.5.2.31 braid_Int ntime

initial number of time intervals

13.5.2.32 braid_PtFcnPhi phi

apply phi function

13.5.2.33 braid_Int print_level

determines amount of output printed to screen (0,1,2)

13.5.2.34 braid_PhiStatus pstatus

status structure passed to user-written phi routines

13.5.2.35 braid_PtFcnRefine refine

(optional) return a refined vector

13.5.2.36 braid_Int* rfactors

refinement factors for finest grid (if any)

13.5.2.37 braid_Real rnorm

residual norm

13.5.2.38 braid_Int rtol

use relative tolerance

13.5.2.39 braid_PtFcnSpatialNorm spatialnorm

Compute norm of a braid_Vector, this is a norm only over space

13.5.2.40 braid_PtFcnSum sum

vector sum

13.5.2.41 braid_Int tnorm

choice of temporal norm

13.5.2.42 braid_Real* tnorm_a

local array of residual norms on a proc's interval, used for inf-norm

13.5.2.43 braid_Real tol

stopping tolerance

13.5.2.44 braid_Real tstart

start time

13.5.2.45 braid_Real tstop

stop time

The documentation for this struct was generated from the following file:

- [_braid.h](#)

13.6 _braid_Grid Struct Reference**Data Fields**

- [braid_Int level](#)
- [braid_Int ilower](#)
- [braid_Int iupper](#)
- [braid_Int clower](#)

- [braid_Int copper](#)
- [braid_Int gupper](#)
- [braid_Int cfactor](#)
- [braid_Int ncpoints](#)
- [braid_Vector * ua](#)
- [braid_Real * ta](#)
- [braid_Vector * va](#)
- [braid_Vector * wa](#)
- [braid_Int recv_index](#)
- [braid_Int send_index](#)
- [_braid_CommHandle * recv_handle](#)
- [_braid_CommHandle * send_handle](#)
- [braid_Vector * ua_alloc](#)
- [braid_Real * ta_alloc](#)
- [braid_Vector * va_alloc](#)
- [braid_Vector * wa_alloc](#)

13.6.1 Detailed Description

XBraid Grid structure for a certain time level

Holds all the information for a processor related to the temporal grid at this level.

13.6.2 Field Documentation

13.6.2.1 **braid_Int cfactor**

coarsening factor

13.6.2.2 **braid_Int clower**

smallest C point index

13.6.2.3 **braid_Int copper**

largest C point index

13.6.2.4 **braid_Int gupper**

global size of the grid

13.6.2.5 **braid_Int ilower**

smallest time index at this level

13.6.2.6 **braid_Int iupper**

largest time index at this level

13.6.2.7 **braid_Int level**

Level that grid is on

13.6.2.8 braid_Int ncpoints

number of C points

13.6.2.9 _braid_CommHandle* recv_handle

Handle for nonblocking receives of braid_Vectors

13.6.2.10 braid_Int recv_index

-1 means no receive

13.6.2.11 _braid_CommHandle* send_handle

Handle for nonblocking sends of braid_Vectors

13.6.2.12 braid_Int send_index

-1 means no send

13.6.2.13 braid_Real* ta

time values (all points)

13.6.2.14 braid_Real* ta_alloc

original memory allocation for ta

13.6.2.15 braid_Vector* ua

unknown vectors (C-points only)

13.6.2.16 braid_Vector* ua_alloc

original memory allocation for ua

13.6.2.17 braid_Vector* va

restricted unknown vectors (all points, NULL on level 0)

13.6.2.18 braid_Vector* va_alloc

original memory allocation for va

13.6.2.19 braid_Vector* wa

rhs vectors f-v (all points, NULL on level 0)

13.6.2.20 braid_Vector* wa_alloc

original memory allocation for wa

The documentation for this struct was generated from the following file:

- [_braid.h](#)

13.7 `_braid_PhiStatus` Struct Reference

Data Fields

- [braid_Real tstart](#)
- [braid_Real tstop](#)
- [braid_Real accuracy](#)
- [braid_Int rfactor](#)

13.7.1 Detailed Description

The user's phi routine routine will receive a `PhiStatus` structure, which defines the status of XBraid at the given instant for phi evaluation on some level during a run. The user accesses it through `braid_PhiStatusGet**()` functions.

13.7.2 Field Documentation

13.7.2.1 `braid_Real accuracy`

advanced option allowing variable accuracy for implicit phi

13.7.2.2 `braid_Int rfactor`

if set by user, allows for subdivision of this interval for better time accuracy

13.7.2.3 `braid_Real tstart`

current time value

13.7.2.4 `braid_Real tstop`

time value to evolve towards, time value to the right of tstart

The documentation for this struct was generated from the following file:

- [braid_status.h](#)

14 File Documentation

14.1 `_braid.h` File Reference

Data Structures

- [struct `_braid_CommHandle`](#)
- [struct `_braid_AccuracyHandle`](#)
- [struct `_braid_Grid`](#)
- [struct `_braid_Core`](#)

Macros

- [#define `_braid_CommHandleElt`](#)(handle, elt) ((handle) -> elt)
- [#define `_braid_GridElt`](#)(grid, elt) ((grid) -> elt)

- `#define _braid_CoreElt(core, elt) ((core) -> elt)`
- `#define _braid_CoreFcn(core, fcn) (*(core) -> fcn)`
- `#define _braid_MapFineToCoarse(findex, cfactor, cindex) (cindex = (findex)/(cfactor))`
- `#define _braid_MapCoarseToFine(cindex, cfactor, findex) (findex = (cindex)*(cfactor))`
- `#define _braid_IsFPoint(index, cfactor) ((index)%(cfactor))`
- `#define _braid_IsCPoint(index, cfactor) (!_braid_IsFPoint(index, cfactor))`

Functions

- `braid_Int _braid_GetDistribution (braid_Core core, braid_Int *ilower_ptr, braid_Int *iupper_ptr)`
- `braid_Int _braid_GetProc (braid_Core core, braid_Int level, braid_Int index, braid_Int *proc_ptr)`
- `braid_Int _braid_CommRecvInit (braid_Core core, braid_Int level, braid_Int index, braid_Vector *vector_ptr, _braid_CommHandle **handle_ptr)`
- `braid_Int _braid_CommSendInit (braid_Core core, braid_Int level, braid_Int index, braid_Vector vector, _braid_CommHandle **handle_ptr)`
- `braid_Int _braid_CommWait (braid_Core core, _braid_CommHandle **handle_ptr)`
- `braid_Int _braid_UCommInit (braid_Core core, braid_Int level)`
- `braid_Int _braid_UCommInitF (braid_Core core, braid_Int level)`
- `braid_Int _braid_UCommWait (braid_Core core, braid_Int level)`
- `braid_Int _braid_UGetInterval (braid_Core core, braid_Int level, braid_Int interval_index, braid_Int *flo_ptr, braid_Int *fhi_ptr, braid_Int *ci_ptr)`
- `braid_Int _braid_UGetVectorRef (braid_Core core, braid_Int level, braid_Int index, braid_Vector *u_ptr)`
- `braid_Int _braid_USetVectorRef (braid_Core core, braid_Int level, braid_Int index, braid_Vector u)`
- `braid_Int _braid_UGetVector (braid_Core core, braid_Int level, braid_Int index, braid_Vector *u_ptr)`
- `braid_Int _braid_USetVector (braid_Core core, braid_Int level, braid_Int index, braid_Vector u)`
- `braid_Int _braid_UAccessVector (braid_Core core, braid_Int level, braid_Int index, braid_Real accuracy, braid_Vector u, braid_Int *rfactor)`
- `braid_Int _braid_Step (braid_Core core, braid_Int level, braid_Int index, braid_Real accuracy, braid_Vector u)`
- `braid_Int _braid_Coarsen (braid_Core core, braid_Int level, braid_Int f_index, braid_Int c_index, braid_Vector fvector, braid_Vector *cvector)`
- `braid_Int _braid_Refine (braid_Core core, braid_Int level, braid_Int f_index, braid_Int c_index, braid_Vector cvector, braid_Vector *fvector)`
- `braid_Int _braid_GridInit (braid_Core core, braid_Int level, braid_Int ilower, braid_Int iupper, _braid_Grid **grid_ptr)`
- `braid_Int _braid_GridDestroy (braid_Core core, _braid_Grid *grid)`
- `braid_Int _braid_InitGuess (braid_Core core, braid_Int level)`
- `braid_Int _braid_FCR Relax (braid_Core core, braid_Int level)`
- `braid_Int _braid_FR restrict (braid_Core core, braid_Int level, braid_Int iter, braid_Real *rnorm_ptr)`
- `braid_Int _braid_FLinter p (braid_Core core, braid_Int level, braid_Int iter, braid_Real rnorm)`
- `braid_Int _braid_FRefine (braid_Core core, braid_Int *refined_ptr)`
- `braid_Int _braid_FAccess (braid_Core core, braid_Real rnorm, braid_Int iter, braid_Int level, braid_Int done)`
- `braid_Int _braid_InitHierarchy (braid_Core core, _braid_Grid *fine_grid)`

Variables

- `FILE * _braid_printfile`

14.1.1 Detailed Description

Define headers for developer routines. This file contains the headers for developer routines.

14.1.2 Macro Definition Documentation

14.1.2.1 `#define _braid_CommHandleElt(handle, elt) ((handle)->elt)`

Accessor for `_braid_CommHandle` attributes

14.1.2.2 `#define _braid_CoreElt(core, elt) ((core)->elt)`

Accessor for `_braid_Core` attributes

14.1.2.3 `#define _braid_CoreFcn(core, fcn) (*(core)->fcn)`

Accessor for `_braid_Core` functions

14.1.2.4 `#define _braid_GridElt(grid, elt) ((grid)->elt)`

Accessor for `_braid_Grid` attributes

14.1.2.5 `#define _braid_IsCPoint(index, cfactor) (!_braid_IsFPoint(index, cfactor))`

Boolean, returns whether a time index is an C-point

14.1.2.6 `#define _braid_IsFPoint(index, cfactor) ((index)%(cfactor))`

Boolean, returns whether a time index is an F-point

14.1.2.7 `#define _braid_MapCoarseToFine(cindex, cfactor, findex) (findex = (cindex)*(cfactor))`

Map a coarse time index to a fine time index, assumes a uniform coarsening factor.

14.1.2.8 `#define _braid_MapFineToCoarse(findex, cfactor, cindex) (cindex = (findex)/(cfactor))`

Map a fine time index to a coarse time index, assumes a uniform coarsening factor.

14.1.3 Function Documentation

14.1.3.1 `braid_Int _braid_Coarsen (braid_Core core, braid_Int level, braid_Int f_index, braid_Int c_index, braid_Vector fvector, braid_Vector * cvector)`

Coarsen in space on *level* by calling the user's coarsen function. The vector corresponding to the time step index *f_index* on the fine grid is coarsened to the time step index *c_index* on the coarse grid. The output goes in *cvector* and the input vector is *fvector*.

14.1.3.2 `braid_Int _braid_CommRecvInit (braid_Core core, braid_Int level, braid_Int index, braid_Vector * vector_ptr, _braid_CommHandle ** handle_ptr)`

Initialize a receive to go into *vector_ptr* for the given time *index* on *level*. Also return a comm handle *handle_ptr* for querying later, to see if the receive has occurred.

14.1.3.3 `braid_Int _braid_CommSendInit (braid_Core core, braid_Int level, braid_Int index, braid_Vector vector, _braid_CommHandle ** handle_ptr)`

Initialize a send of *vector* for the given time *index* on *level*. Also return a comm handle *handle_ptr* for querying later, to see if the send has occurred.

14.1.3.4 **braid_Int _braid_CommWait (braid_Core core, _braid_CommHandle ** handle_ptr)**

Block on the comm handle *handle_ptr* until the MPI operation (send or recv) has completed

14.1.3.5 **braid_Int _braid_FAccess (braid_Core core, braid_Real rnorm, braid_Int iter, braid_Int level, braid_Int done)**

Call the user's access function in order to give access to XBraid and the current vector at grid *level* and iteration **iter*. Most commonly, this lets the user write solutions to screen, disk, etc... The quantity *rnorm* denotes the last computed residual norm, and *done* is a boolean indicating whether XBraid has finished iterating and this is the last Access call.

14.1.3.6 **braid_Int _braid_FCRelax (braid_Core core, braid_Int level)**

Do nu sweeps of F-then-C relaxation on *level*

14.1.3.7 **braid_Int _braid_FInterp (braid_Core core, braid_Int level, braid_Int iter, braid_Real rnorm)**

F-Relax on *level* and interpolate to *level-1*

The output is set in the *braid_Grid* in *core*, so that the vector *u* on *level* is created by interpolating from *level+1*.

If the user has set spatial refinement, then this user-defined routine is also called.

Parameters

<i>core</i>	braid_Core (_braid_Core) struct
<i>level</i>	interp from level to level+1
<i>iter</i>	current iteration number (for user info)
<i>rnorm</i>	residual norm (if level 0)

14.1.3.8 **braid_Int _braid_FRefine (braid_Core core, braid_Int * refined_ptr)**

Create a new fine grid based on user refinement factor information, then F-relax and interpolate to the new fine grid and create a new multigrid hierarchy. In general, this will require load re-balancing as well.

RDF: Todo, routine is unwritten

14.1.3.9 **braid_Int _braid_FRestrict (braid_Core core, braid_Int level, braid_Int iter, braid_Real * rnorm_ptr)**

F-Relax on *level* and then restrict to *level+1*

The output is set in the *braid_Grid* in *core*, so that the restricted vectors *va* and *wa* will be created, representing *level+1* versions of the unknown and rhs vectors.

If the user has set spatial coarsening, then this user-defined routine is also called.

If *level==0*, then *rnorm_ptr* will contain the residual norm.

Parameters

<i>core</i>	braid_Core (_braid_Core) struct
<i>level</i>	restrict from level to level+1
<i>iter</i>	current iteration number (for user info)
<i>rnorm_ptr</i>	pointer to residual norm (if level 0)

14.1.3.10 **braid_Int _braid_GetDistribution (braid_Core core, braid_Int * ilower_ptr, braid_Int * iupper_ptr)**

Determine processor distribution. This must agree with *GetProc()*. For the processor rank calling this function, it returns the smallest and largest time indices (*ilower_ptr* and *iupper_ptr*) that belong to that processor (the indices may * be F or C points).

14.1.3.11 `braid_Int _braid_GetProc (braid_Core core, braid_Int level, braid_Int index, braid_Int * proc_ptr)`

Return the processor number in *proc_ptr* on which the time step *index* lives for the given *level*. * Returns -1 if *index* is out of range

14.1.3.12 `braid_Int _braid_GridDestroy (braid_Core core, _braid_Grid * grid)`

Destroy an XBraid *grid*

14.1.3.13 `braid_Int _braid_GridInit (braid_Core core, braid_Int level, braid_Int ilower, braid_Int iupper, _braid_Grid ** grid_ptr)`

Create a new grid object *grid_ptr* in core at *level*

ilower and *iupper* correspond to the lower and upper time index values for this processor on this grid.

14.1.3.14 `braid_Int _braid_InitGuess (braid_Core core, braid_Int level)`

Set initial guess at C-points on *level*

14.1.3.15 `braid_Int _braid_InitHierarchy (braid_Core core, _braid_Grid * fine_grid)`

Initialize (and re-initialize) hierarchy

14.1.3.16 `braid_Int _braid_Phi (braid_Core core, braid_Int level, braid_Int index, braid_Real accuracy, braid_Vector u, braid_Int * rfactor)`

Apply Phi to the vector *u*

This is the vector corresponding to the time step *index* on *level*. *accuracy* is a user set variable to allow for tuning the accuracy of spatial solves for implicit stepping. And, *rfactor* allows the user to subdivide time intervals for accuracy purposes.

14.1.3.17 `braid_Int _braid_Refine (braid_Core core, braid_Int level, braid_Int f_index, braid_Int c_index, braid_Vector cvector, braid_Vector * fvector)`

Refine in space on *level* by calling the user's refine function. The vector corresponding to the time step index *c_index* on the coarse grid is refined to the time step index *f_index* on the fine grid. The output goes in *fvector* and the input vector is *cvector*.

14.1.3.18 `braid_Int _braid_Step (braid_Core core, braid_Int level, braid_Int index, braid_Real accuracy, braid_Vector u)`

Integrate one time step at time step *index* to time step *index*+1

14.1.3.19 `braid_Int _braid_UAccessVector (braid_Core core, braid_AccessStatus status, braid_Vector u)`

Call user's access function in order to give access to XBraid and the current vector. Most commonly, this lets the user write *u* to screen, disk, etc... The vector *u* corresponds to time step *index* on *level*. *status* holds state information about the current XBraid iteration, time value, etc...

14.1.3.20 `braid_Int _braid_UCommlnit (braid_Core core, braid_Int level)`

Working on all intervals

At *level*, post a receive for the point to the left of *ilower* (regardless whether *ilower* is F or C). Then, post a send of *iupper* if *iupper* is a C point.

14.1.3.21 `braid_Int _braid_UCommInitF (braid_Core core, braid_Int level)`

Working only on F-pt intervals

At *level*, **only** post a receive for the point to the left of *ilower* if *ilower* is an F point. Then, post a send of *iupper* if *iupper* is a C point.

14.1.3.22 `braid_Int _braid_UCommWait (braid_Core core, braid_Int level)`

Finish up communication

On *level*, wait on both the recv and send handles at this level.

14.1.3.23 `braid_Int _braid_UGetInterval (braid_Core core, braid_Int level, braid_Int interval_index, braid_Int * flo_ptr, braid_Int * fhi_ptr, braid_Int * ci_ptr)`

Retrieve the time step indices at this *level* which correspond to the FC interval given by *interval_index*. *ci_ptr* is the time step index for the C point and *flo_ptr* and *fhi_ptr* are the smallest and largest F point indices in this interval. $flo = ci + 1$, and $fhi = ci + coarsening_factor - 1$

14.1.3.24 `braid_Int _braid_UGetVector (braid_Core core, braid_Int level, braid_Int index, braid_Vector * u_ptr)`

Returns the u-vector in *u_ptr* on grid *level* at point *index*. If *index* is my "receive index" (as set by `UCommInit()`, for example), the u-vector will be received from a neighbor processor. If *index* is within my index range and is also a C-point, the saved value of u will be used. A NULL value is returned otherwise.

14.1.3.25 `braid_Int _braid_UGetVectorRef (braid_Core core, braid_Int level, braid_Int index, braid_Vector * u_ptr)`

Returns a reference to the local u-vector in *u_ptr* for the grid *level* at point *index*. Caveat: if *index* is not a C-point and within my index range, NULL is returned.

14.1.3.26 `braid_Int _braid_USetVector (braid_Core core, braid_Int level, braid_Int index, braid_Vector u)`

Sets the u-vector on grid *level* at point *index*. If *index* is my "send index" (as set by `UCommInit()`, for example), a send is initiated to a neighbor processor. If *index* is within my index range and is also a C-point, the value is saved locally.

14.1.3.27 `braid_Int _braid_USetVectorRef (braid_Core core, braid_Int level, braid_Int index, braid_Vector u)`

Stores a reference to the vector *u* on grid *level* at point *index*. If *index* is not a C-point and within this processor's range of time points, then nothing is done.

14.1.4 Variable Documentation

14.1.4.1 `FILE* _braid_printfile`

This is the print file for redirecting stdout for all XBraid screen output

14.2 `braid.h` File Reference

Typedefs

- typedef struct `_braid_App_struct` * `braid_App`
- typedef struct `_braid_Vector_struct` * `braid_Vector`
- typedef `braid_Int`(* `braid_PtFcnPhi`)(`braid_App` app, `braid_Vector` u, `braid_PhiStatus` status)

- typedef `braid_Int`(* `braid_PtFcnInit`)(`braid_App` app, `braid_Real` t, `braid_Vector` *u_ptr)
- typedef `braid_Int`(* `braid_PtFcnClone`)(`braid_App` app, `braid_Vector` u, `braid_Vector` *v_ptr)
- typedef `braid_Int`(* `braid_PtFcnFree`)(`braid_App` app, `braid_Vector` u)
- typedef `braid_Int`(* `braid_PtFcnSum`)(`braid_App` app, `braid_Real` alpha, `braid_Vector` x, `braid_Real` beta, `braid_Vector` y)
- typedef `braid_Int`(* `braid_PtFcnSpatialNorm`)(`braid_App` app, `braid_Vector` u, `braid_Real` *norm_ptr)
- typedef `braid_Int`(* `braid_PtFcnAccess`)(`braid_App` app, `braid_Vector` u, `braid_AccessStatus` status)
- typedef `braid_Int`(* `braid_PtFcnBufSize`)(`braid_App` app, `braid_Int` *size_ptr)
- typedef `braid_Int`(* `braid_PtFcnBufPack`)(`braid_App` app, `braid_Vector` u, void *buffer, `braid_Int` *size_ptr)
- typedef `braid_Int`(* `braid_PtFcnBufUnpack`)(`braid_App` app, void *buffer, `braid_Vector` *u_ptr)
- typedef `braid_Int`(* `braid_PtFcnCoarsen`)(`braid_App` app, `braid_Vector` fu, `braid_Vector` *cu_ptr, `braid_CoarsenRefStatus` status)
- typedef `braid_Int`(* `braid_PtFcnRefine`)(`braid_App` app, `braid_Vector` cu, `braid_Vector` *fu_ptr, `braid_CoarsenRefStatus` status)
- typedef struct `_braid_Core_struct` * `braid_Core`

Functions

- `braid_Int` `braid_Init` (`MPI_Comm` comm_world, `MPI_Comm` comm, `braid_Real` tstart, `braid_Real` tstop, `braid_Int` ntime, `braid_App` app, `braid_PtFcnPhi` phi, `braid_PtFcnInit` init, `braid_PtFcnClone` clone, `braid_PtFcnFree` free, `braid_PtFcnSum` sum, `braid_PtFcnSpatialNorm` spatialnorm, `braid_PtFcnAccess` access, `braid_PtFcnBufSize` bufsize, `braid_PtFcnBufPack` bufpack, `braid_PtFcnBufUnpack` bufunpack, `braid_Core` *core_ptr)
- `braid_Int` `braid_Drive` (`braid_Core` core)
- `braid_Int` `braid_Destroy` (`braid_Core` core)
- `braid_Int` `braid_PrintStats` (`braid_Core` core)
- `braid_Int` `braid_SetLoosexTol` (`braid_Core` core, `braid_Int` level, `braid_Real` loose_tol)
- `braid_Int` `braid_SetTightxTol` (`braid_Core` core, `braid_Int` level, `braid_Real` tight_tol)
- `braid_Int` `braid_SetMaxLevels` (`braid_Core` core, `braid_Int` max_levels)
- `braid_Int` `braid_SetMinCoarse` (`braid_Core` core, `braid_Int` min_coarse)
- `braid_Int` `braid_SetAbsTol` (`braid_Core` core, `braid_Real` atol)
- `braid_Int` `braid_SetRelTol` (`braid_Core` core, `braid_Real` rtol)
- `braid_Int` `braid_SetNRelax` (`braid_Core` core, `braid_Int` level, `braid_Int` nrelax)
- `braid_Int` `braid_SetCFactor` (`braid_Core` core, `braid_Int` level, `braid_Int` cfactor)
- `braid_Int` `braid_SetMaxIter` (`braid_Core` core, `braid_Int` max_iter)
- `braid_Int` `braid_SetFMG` (`braid_Core` core)
- `braid_Int` `braid_SetTemporalNorm` (`braid_Core` core, `braid_Int` tnorm)
- `braid_Int` `braid_SetNFMGVcyc` (`braid_Core` core, `braid_Int` nfmg_Vcyc)
- `braid_Int` `braid_SetSpatialCoarsen` (`braid_Core` core, `braid_PtFcnCoarsen` coarsen)
- `braid_Int` `braid_SetSpatialRefine` (`braid_Core` core, `braid_PtFcnRefine` refine)
- `braid_Int` `braid_SetPrintLevel` (`braid_Core` core, `braid_Int` print_level)
- `braid_Int` `braid_SetPrintFile` (`braid_Core` core, const char *printfile_name)
- `braid_Int` `braid_SetAccessLevel` (`braid_Core` core, `braid_Int` access_level)
- `braid_Int` `braid_SplitCommworld` (const `MPI_Comm` *comm_world, `braid_Int` px, `MPI_Comm` *comm_x, `MPI_Comm` *comm_t)
- `braid_Int` `braid_GetNumIter` (`braid_Core` core, `braid_Int` *niter_ptr)
- `braid_Int` `braid_GetRNorm` (`braid_Core` core, `braid_Real` *rnorm_ptr)

14.2.1 Detailed Description

Define headers for user interface routines. This file contains routines used to allow the user to initialize, run and get and set a XBraid solver.

14.3 braid_defs.h File Reference

Macros

- `#define _braid_TAlloc(type, count) ((type *)malloc((size_t)(sizeof(type) * (count))))`
- `#define _braid_CTAlloc(type, count) ((type *)calloc((size_t)(count), (size_t)sizeof(type)))`
- `#define _braid_TReAlloc(ptr, type, count) ((type *)realloc((char *)ptr, (size_t)(sizeof(type) * (count))))`
- `#define _braid_TFree(ptr) (free((char *)ptr), ptr = NULL)`

Typedefs

- `typedef int braid_Int`
- `typedef double braid_Real`

Variables

- `braid_Int braid_error_flag`

14.3.1 Detailed Description

Definitions of types, error flags, etc...

14.3.2 Macro Definition Documentation

14.3.2.1 `#define _braid_CTAlloc(type, count) ((type *)calloc((size_t)(count), (size_t)sizeof(type)))`

Allocation macro

14.3.2.2 `#define _braid_TAlloc(type, count) ((type *)malloc((size_t)(sizeof(type) * (count))))`

Allocation macro

14.3.2.3 `#define _braid_TFree(ptr) (free((char *)ptr), ptr = NULL)`

Free memory macro

14.3.2.4 `#define _braid_TReAlloc(ptr, type, count) ((type *)realloc((char *)ptr, (size_t)(sizeof(type) * (count))))`

Re-allocation macro

14.3.3 Typedef Documentation

14.3.3.1 `typedef int braid_Int`

Defines integer type

14.3.3.2 `typedef double braid_Real`

Defines floating point type

14.3.4 Variable Documentation

14.3.4.1 braid_Int _braid_error_flag

This is the global XBraid error flag. If it is ever nonzero, an error has occurred.

14.4 braid_status.h File Reference

Data Structures

- struct [_braid_AccessStatus](#)
- struct [_braid_CoarsenRefStatus](#)
- struct [_braid_PhiStatus](#)

Macros

- `#define _braid_StatusElt(status, elt) ((status) -> elt)`

Typedefs

- typedef struct [_braid_AccessStatus_struct](#) * [braid_AccessStatus](#)
- typedef struct [_braid_CoarsenRefStatus_struct](#) * [braid_CoarsenRefStatus](#)
- typedef struct [_braid_PhiStatus_struct](#) * [braid_PhiStatus](#)

Functions

- [braid_Int _braid_AccessStatusInit](#) ([braid_Real](#) t, [braid_Real](#) rnorm, [braid_Int](#) iter, [braid_Int](#) level, [braid_Int](#) done, [braid_Int](#) wrapper_test, [braid_AccessStatus](#) status)
- [braid_Int _braid_AccessStatusDestroy](#) ([braid_AccessStatus](#) status)
- [braid_Int braid_AccessStatusGetT](#) ([braid_AccessStatus](#) status, [braid_Real](#) *t_ptr)
- [braid_Int braid_AccessStatusGetResidual](#) ([braid_AccessStatus](#) status, [braid_Real](#) *rnorm_ptr)
- [braid_Int braid_AccessStatusGetIter](#) ([braid_AccessStatus](#) status, [braid_Int](#) *iter_ptr)
- [braid_Int braid_AccessStatusGetLevel](#) ([braid_AccessStatus](#) status, [braid_Int](#) *level_ptr)
- [braid_Int braid_AccessStatusGetDone](#) ([braid_AccessStatus](#) status, [braid_Int](#) *done_ptr)
- [braid_Int braid_AccessStatusGetWrapperTest](#) ([braid_AccessStatus](#) status, [braid_Int](#) *wtest_ptr)
- [braid_Int braid_AccessStatusGetTILD](#) ([braid_AccessStatus](#) status, [braid_Real](#) *t_ptr, [braid_Int](#) *iter_ptr, [braid_Int](#) *level_ptr, [braid_Int](#) *done_ptr)
- [braid_Int _braid_CoarsenRefStatusInit](#) ([braid_Real](#) tstart, [braid_Real](#) f_tprior, [braid_Real](#) f_tstop, [braid_Real](#) c_tprior, [braid_Real](#) c_tstop, [braid_CoarsenRefStatus](#) status)
- [braid_Int _braid_CoarsenRefStatusDestroy](#) ([braid_CoarsenRefStatus](#) status)
- [braid_Int braid_CoarsenRefStatusGetTstart](#) ([braid_CoarsenRefStatus](#) status, [braid_Real](#) *tstart_ptr)
- [braid_Int braid_CoarsenRefStatusGetFTstop](#) ([braid_CoarsenRefStatus](#) status, [braid_Real](#) *f_tstop_ptr)
- [braid_Int braid_CoarsenRefStatusGetFTprior](#) ([braid_CoarsenRefStatus](#) status, [braid_Real](#) *f_tprior_ptr)
- [braid_Int braid_CoarsenRefStatusGetCTstop](#) ([braid_CoarsenRefStatus](#) status, [braid_Real](#) *c_tstop_ptr)
- [braid_Int braid_CoarsenRefStatusGetCTprior](#) ([braid_CoarsenRefStatus](#) status, [braid_Real](#) *c_tprior_ptr)
- [braid_Int braid_CoarsenRefStatusGetTpriorTstop](#) ([braid_CoarsenRefStatus](#) status, [braid_Real](#) *tstart_ptr, [braid_Real](#) *f_tprior_ptr, [braid_Real](#) *f_tstop_ptr, [braid_Real](#) *c_tprior_ptr, [braid_Real](#) *c_tstop_ptr)

- [braid_Int _braid_PhiStatusInit](#) ([braid_Real](#) tstart, [braid_Real](#) tstop, [braid_Real](#) accuracy, [braid_PhiStatus](#) status)
- [braid_Int _braid_PhiStatusDestroy](#) ([braid_PhiStatus](#) status)
- [braid_Int braid_PhiStatusGetTstart](#) ([braid_PhiStatus](#) status, [braid_Real](#) *tstart_ptr)
- [braid_Int braid_PhiStatusGetTstop](#) ([braid_PhiStatus](#) status, [braid_Real](#) *tstop_ptr)
- [braid_Int braid_PhiStatusGetAccuracy](#) ([braid_PhiStatus](#) status, [braid_Real](#) *accuracy_ptr)
- [braid_Int braid_PhiStatusSetRFactor](#) ([braid_PhiStatus](#) status, [braid_Real](#) rfactor)
- [braid_Int braid_PhiStatusGetTstartTstop](#) ([braid_PhiStatus](#) status, [braid_Real](#) *tstart_ptr, [braid_Real](#) *tstop_ptr)

14.4.1 Detailed Description

Define headers for XBraid status structures, status get/set routines and status create/destroy routines.

14.5 braid_test.h File Reference

Functions

- [braid_Int braid_TestInitAccess](#) ([braid_App](#) app, [MPI_Comm](#) comm_x, [FILE](#) *fp, [braid_Real](#) t, [braid_PtFcnInit](#) init, [braid_PtFcnAccess](#) access, [braid_PtFcnFree](#) free)
- [braid_Int braid_TestClone](#) ([braid_App](#) app, [MPI_Comm](#) comm_x, [FILE](#) *fp, [braid_Real](#) t, [braid_PtFcnInit](#) init, [braid_PtFcnAccess](#) access, [braid_PtFcnFree](#) free, [braid_PtFcnClone](#) clone)
- [braid_Int braid_TestSum](#) ([braid_App](#) app, [MPI_Comm](#) comm_x, [FILE](#) *fp, [braid_Real](#) t, [braid_PtFcnInit](#) init, [braid_PtFcnAccess](#) access, [braid_PtFcnFree](#) free, [braid_PtFcnClone](#) clone, [braid_PtFcnSum](#) sum)
- [braid_Int braid_TestSpatialNorm](#) ([braid_App](#) app, [MPI_Comm](#) comm_x, [FILE](#) *fp, [braid_Real](#) t, [braid_PtFcnInit](#) init, [braid_PtFcnFree](#) free, [braid_PtFcnClone](#) clone, [braid_PtFcnSum](#) sum, [braid_PtFcnSpatialNorm](#) spatialnorm)
- [braid_Int braid_TestBuf](#) ([braid_App](#) app, [MPI_Comm](#) comm_x, [FILE](#) *fp, [braid_Real](#) t, [braid_PtFcnInit](#) init, [braid_PtFcnFree](#) free, [braid_PtFcnSum](#) sum, [braid_PtFcnSpatialNorm](#) spatialnorm, [braid_PtFcnBufSize](#) bufsize, [braid_PtFcnBufPack](#) bufpack, [braid_PtFcnBufUnpack](#) bufunpack)
- [braid_Int braid_TestCoarsenRefine](#) ([braid_App](#) app, [MPI_Comm](#) comm_x, [FILE](#) *fp, [braid_Real](#) t, [braid_Real](#) fdt, [braid_Real](#) cdt, [braid_PtFcnInit](#) init, [braid_PtFcnAccess](#) access, [braid_PtFcnFree](#) free, [braid_PtFcnClone](#) clone, [braid_PtFcnSum](#) sum, [braid_PtFcnSpatialNorm](#) spatialnorm, [braid_PtFcnCoarsen](#) coarsen, [braid_PtFcnRefine](#) refine)
- [braid_Int braid_TestAll](#) ([braid_App](#) app, [MPI_Comm](#) comm_x, [FILE](#) *fp, [braid_Real](#) t, [braid_Real](#) fdt, [braid_Real](#) cdt, [braid_PtFcnInit](#) init, [braid_PtFcnFree](#) free, [braid_PtFcnClone](#) clone, [braid_PtFcnSum](#) sum, [braid_PtFcnSpatialNorm](#) spatialnorm, [braid_PtFcnBufSize](#) bufsize, [braid_PtFcnBufPack](#) bufpack, [braid_PtFcnBufUnpack](#) bufunpack, [braid_PtFcnCoarsen](#) coarsen, [braid_PtFcnRefine](#) refine)

14.5.1 Detailed Description

Define headers for XBraid test routines. This file contains routines used to test a user's XBraid wrapper routines one-by-one.

14.6 util.h File Reference

Functions

- [braid_Int _braid_ProjectInterval](#) ([braid_Int](#) ilower, [braid_Int](#) iupper, [braid_Int](#) index, [braid_Int](#) stride, [braid_Int](#) *pilower, [braid_Int](#) *piupper)
- [braid_Int _braid_SetAccuracy](#) ([braid_Real](#) rnorm, [braid_Real](#) loose_tol, [braid_Real](#) tight_tol, [braid_Real](#) old-Accuracy, [braid_Real](#) tol, [braid_Real](#) *paccuracy)
- [braid_Int _braid_printf](#) (const char *format,...)

- [braid_Int_braid_ParPrintfFlush](#) (FILE *file, [braid_Int](#) myid, char *message,...)
- [braid_Int_braid_Max](#) ([braid_Real](#) *array, [braid_Int](#) size, [braid_Real](#) *max_val)

14.6.1 Detailed Description

Define headers for utility routines. This file contains the headers for utility routines. Essentially, if a routine does not take [braid_Core](#) (or other XBraid specific structs) as an argument, then it's a utility routine.

14.6.2 Function Documentation

14.6.2.1 [braid_Int_braid_Max](#) ([braid_Real](#) * array, [braid_Int](#) size, [braid_Real](#) * max_val)

This function finds the maximum value in a [braid_Real](#) array

14.6.2.2 [braid_Int_braid_ParPrintfFlush](#) (FILE * file, [braid_Int](#) myid, char * message, ...)

This is a function that allows for "sane" printing of information in parallel. Currently, only myid = 0 prints, but this can be updated as needs change.

The string *message* is printed and can be multiple parameters in the standard * C-format, like

```
message = '%1.2e is a format string', 1.24
```

14.6.2.3 [braid_Int_braid_printf](#) (const char * format, ...)

If set, print to [_braid_printfile](#) and then flush. Else print to standard out.

The string *format* can be multiple parameters in the standard * C-format, like

```
format = '%1.2e is a format string', 1.24
```

14.6.2.4 [braid_Int_braid_ProjectInterval](#) ([braid_Int](#) ilower, [braid_Int](#) iupper, [braid_Int](#) index, [braid_Int](#) stride, [braid_Int](#) * pilower, [braid_Int](#) * piupper)

Project an interval onto a strided index space that contains the index 'index' and has stride 'stride'. An empty projection is represented by ilower > iupper.

14.6.2.5 [braid_Int_braid_SetAccuracy](#) ([braid_Real](#) rnorm, [braid_Real](#) loose_tol, [braid_Real](#) tight_tol, [braid_Real](#) oldAccuracy, [braid_Real](#) tol, [braid_Real](#) * paccuracy)

Determine the accuracy used for the spatial solves based on the ratio of the current residual norm and the stopping tolerance.

Index

- [_braid.h, 61](#)
 - [_braid_Coarsen, 63](#)
 - [_braid_CommHandleElt, 63](#)
 - [_braid_CommRecvInit, 63](#)
 - [_braid_CommSendInit, 63](#)
 - [_braid_CommWait, 63](#)
 - [_braid_CoreElt, 63](#)
 - [_braid_CoreFcn, 63](#)
 - [_braid_FAccess, 64](#)
 - [_braid_FCRelax, 64](#)
 - [_braid_FInterp, 64](#)
 - [_braid_FRefine, 64](#)
 - [_braid_FRestrict, 64](#)
 - [_braid_GetDistribution, 64](#)
 - [_braid_GetProc, 64](#)
 - [_braid_GridDestroy, 65](#)
 - [_braid_GridElt, 63](#)
 - [_braid_GridInit, 65](#)
 - [_braid_InitGuess, 65](#)
 - [_braid_InitHierarchy, 65](#)
 - [_braid_IsCPoint, 63](#)
 - [_braid_IsFPoint, 63](#)
 - [_braid_MapCoarseToFine, 63](#)
 - [_braid_MapFineToCoarse, 63](#)
 - [_braid_Phi, 65](#)
 - [_braid_Refine, 65](#)
 - [_braid_Step, 65](#)
 - [_braid_UAccessVector, 65](#)
 - [_braid_UCommInit, 65](#)
 - [_braid_UCommInitF, 65](#)
 - [_braid_UCommWait, 66](#)
 - [_braid_UGetInterval, 66](#)
 - [_braid_UGetVector, 66](#)
 - [_braid_UGetVectorRef, 66](#)
 - [_braid_USetVector, 66](#)
 - [_braid_USetVectorRef, 66](#)
 - [_braid_printfile, 66](#)
- [_braid_AccessStatus, 51](#)
 - [done, 51](#)
 - [iter, 51](#)
 - [level, 51](#)
 - [rnorm, 51](#)
 - [t, 51](#)
 - [wrapper_test, 51](#)
- [_braid_AccessStatusDestroy](#)
 - [XBraid status routines, 39](#)
- [_braid_AccessStatusInit](#)
 - [XBraid status routines, 39](#)
- [_braid_AccuracyHandle, 51](#)
 - [loose, 52](#)
 - [matchF, 52](#)
 - [old_value, 52](#)
 - [tight, 52](#)
 - [tight_used, 52](#)
 - [value, 52](#)
- [_braid_CTAlloc](#)
 - [braid_defs.h, 68](#)
- [_braid_Coarsen](#)
 - [_braid.h, 63](#)
- [_braid_CoarsenRefStatus, 52](#)
 - [c_tprior, 53](#)
 - [c_tstop, 53](#)
 - [f_tprior, 53](#)
 - [f_tstop, 53](#)
 - [tstart, 53](#)
- [_braid_CoarsenRefStatusDestroy](#)
 - [XBraid status routines, 39](#)
- [_braid_CoarsenRefStatusInit](#)
 - [XBraid status routines, 39](#)
- [_braid_CommHandle, 53](#)
 - [buffer, 53](#)
 - [num_requests, 54](#)
 - [request_type, 54](#)
 - [requests, 54](#)
 - [status, 54](#)
 - [vector_ptr, 54](#)
- [_braid_CommHandleElt](#)
 - [_braid.h, 63](#)
- [_braid_CommRecvInit](#)
 - [_braid.h, 63](#)
- [_braid_CommSendInit](#)
 - [_braid.h, 63](#)
- [_braid_CommWait](#)
 - [_braid.h, 63](#)
- [_braid_Core, 54](#)
 - [access, 55](#)
 - [access_level, 55](#)
 - [accuracy, 55](#)
 - [app, 55](#)
 - [astatus, 55](#)
 - [bufpack, 55](#)
 - [bufsize, 56](#)
 - [bufunpack, 56](#)
 - [cfactors, 56](#)
 - [cfdefault, 56](#)
 - [clone, 56](#)
 - [coarsen, 56](#)
 - [comm, 56](#)
 - [comm_world, 56](#)
 - [cstatus, 56](#)
 - [fmg, 56](#)
 - [free, 56](#)

- globaltime, 56
- grids, 56
- gupper, 56
- init, 56
- localtime, 57
- max_iter, 57
- max_levels, 57
- min_coarse, 57
- nfmng_Vcyc, 57
- niter, 57
- nlevels, 57
- nrdefault, 57
- nrels, 57
- ntime, 57
- phi, 57
- print_level, 57
- pstatus, 57
- refine, 57
- rfactors, 58
- rnorm, 58
- rtol, 58
- spatialnorm, 58
- sum, 58
- tnorm, 58
- tnorm_a, 58
- tol, 58
- tstart, 58
- tstop, 58
- _braid_CoreElt
 - _braid.h, 63
- _braid_CoreFcn
 - _braid.h, 63
- _braid_FAccess
 - _braid.h, 64
- _braid_FCRelax
 - _braid.h, 64
- _braid_FInterp
 - _braid.h, 64
- _braid_FRefine
 - _braid.h, 64
- _braid_FRestrict
 - _braid.h, 64
- _braid_GetDistribution
 - _braid.h, 64
- _braid_GetProc
 - _braid.h, 64
- _braid_Grid, 58
 - cfactor, 59
 - clower, 59
 - cupper, 59
 - gupper, 59
 - ilower, 59
 - iupper, 59
 - level, 59
 - ncpoints, 59
 - recv_handle, 60
 - recv_index, 60
 - send_handle, 60
 - send_index, 60
 - ta, 60
 - ta_alloc, 60
 - ua, 60
 - ua_alloc, 60
 - va, 60
 - va_alloc, 60
 - wa, 60
 - wa_alloc, 60
- _braid_GridDestroy
 - _braid.h, 65
- _braid_GridElt
 - _braid.h, 63
- _braid_GridInit
 - _braid.h, 65
- _braid_InitGuess
 - _braid.h, 65
- _braid_InitHierarchy
 - _braid.h, 65
- _braid_IsCPoint
 - _braid.h, 63
- _braid_IsFPoint
 - _braid.h, 63
- _braid_MapCoarseToFine
 - _braid.h, 63
- _braid_MapFineToCoarse
 - _braid.h, 63
- _braid_Max
 - util.h, 71
- _braid_ParFprintfFlush
 - util.h, 71
- _braid_Phi
 - _braid.h, 65
- _braid_PhiStatus, 61
 - accuracy, 61
 - rfactor, 61
 - tstart, 61
 - tstop, 61
- _braid_PhiStatusDestroy
 - XBraid status routines, 41
- _braid_PhiStatusInit
 - XBraid status routines, 41
- _braid_ProjectInterval
 - util.h, 71
- _braid_Refine
 - _braid.h, 65
- _braid_SetAccuracy
 - util.h, 71
- _braid_StatusElt
 - XBraid status routines, 39

- `_braid_Step`
 - `_braid.h`, 65
- `_braid_TAlloc`
 - `braid_defs.h`, 68
- `_braid_TFree`
 - `braid_defs.h`, 68
- `_braid_TReAlloc`
 - `braid_defs.h`, 68
- `_braid_UAccessVector`
 - `_braid.h`, 65
- `_braid_UCommInit`
 - `_braid.h`, 65
- `_braid_UCommInitF`
 - `_braid.h`, 65
- `_braid_UCommWait`
 - `_braid.h`, 66
- `_braid_UGetInterval`
 - `_braid.h`, 66
- `_braid_UGetVector`
 - `_braid.h`, 66
- `_braid_UGetVectorRef`
 - `_braid.h`, 66
- `_braid_USetVector`
 - `_braid.h`, 66
- `_braid_USetVectorRef`
 - `_braid.h`, 66
- `_braid_error_flag`
 - `braid_defs.h`, 69
- `_braid_printf`
 - `util.h`, 71
- `_braid_printfile`
 - `_braid.h`, 66
- `access`
 - `_braid_Core`, 55
- `access_level`
 - `_braid_Core`, 55
- `accuracy`
 - `_braid_Core`, 55
 - `_braid_PhiStatus`, 61
- `app`
 - `_braid_Core`, 55
- `astatus`
 - `_braid_Core`, 55
- `braid.h`, 66
- `braid_AccessStatus`
 - XBraid status routines, 39
- `braid_AccessStatusGetDone`
 - XBraid status routines, 41
- `braid_AccessStatusGetIter`
 - XBraid status routines, 41
- `braid_AccessStatusGetLevel`
 - XBraid status routines, 41
- `braid_AccessStatusGetResidual`
 - XBraid status routines, 42
- `braid_AccessStatusGetT`
 - XBraid status routines, 42
- `braid_AccessStatusGetTILD`
 - XBraid status routines, 42
- `braid_AccessStatusGetWrapperTest`
 - XBraid status routines, 42
- `braid_App`
 - User-written routines, 26
- `braid_CoarsenRefStatus`
 - XBraid status routines, 39
- `braid_CoarsenRefStatusGetCTprior`
 - XBraid status routines, 42
- `braid_CoarsenRefStatusGetCTstop`
 - XBraid status routines, 43
- `braid_CoarsenRefStatusGetFTprior`
 - XBraid status routines, 43
- `braid_CoarsenRefStatusGetFTstop`
 - XBraid status routines, 43
- `braid_CoarsenRefStatusGetTpriorTstop`
 - XBraid status routines, 43
- `braid_CoarsenRefStatusGetTstart`
 - XBraid status routines, 43
- `braid_Core`
 - General Interface routines, 30
- `braid_Destroy`
 - General Interface routines, 31
- `braid_Drive`
 - General Interface routines, 32
- `braid_GetNumIter`
 - General Interface routines, 32
- `braid_GetRNorm`
 - General Interface routines, 32
- `braid_Init`
 - General Interface routines, 32
- `braid_Int`
 - `braid_defs.h`, 68
- `braid_PhiStatus`
 - XBraid status routines, 39
- `braid_PhiStatusGetAccuracy`
 - XBraid status routines, 44
- `braid_PhiStatusGetTstart`
 - XBraid status routines, 44
- `braid_PhiStatusGetTstartTstop`
 - XBraid status routines, 44
- `braid_PhiStatusGetTstop`
 - XBraid status routines, 44
- `braid_PhiStatusSetRFactor`
 - XBraid status routines, 44
- `braid_PrintStats`
 - General Interface routines, 33
- `braid_PtFcnAccess`
 - User-written routines, 26

- braid_PtFcnBufPack
 - User-written routines, [26](#)
- braid_PtFcnBufSize
 - User-written routines, [26](#)
- braid_PtFcnBufUnpack
 - User-written routines, [27](#)
- braid_PtFcnClone
 - User-written routines, [27](#)
- braid_PtFcnCoarsen
 - User-written routines, [27](#)
- braid_PtFcnFree
 - User-written routines, [27](#)
- braid_PtFcnInit
 - User-written routines, [27](#)
- braid_PtFcnPhi
 - User-written routines, [27](#)
- braid_PtFcnRefine
 - User-written routines, [27](#)
- braid_PtFcnSpatialNorm
 - User-written routines, [27](#)
- braid_PtFcnSum
 - User-written routines, [28](#)
- braid_Real
 - [braid_defs.h](#), [68](#)
- braid_SetAbsTol
 - General Interface routines, [33](#)
- braid_SetAccessLevel
 - General Interface routines, [33](#)
- braid_SetCFactor
 - General Interface routines, [33](#)
- braid_SetFMG
 - General Interface routines, [34](#)
- braid_SetLoosexTol
 - General Interface routines, [34](#)
- braid_SetMaxIter
 - General Interface routines, [34](#)
- braid_SetMaxLevels
 - General Interface routines, [34](#)
- braid_SetMinCoarse
 - General Interface routines, [34](#)
- braid_SetNFMGVcyc
 - General Interface routines, [34](#)
- braid_SetNRelax
 - General Interface routines, [35](#)
- braid_SetPrintFile
 - General Interface routines, [35](#)
- braid_SetPrintLevel
 - General Interface routines, [35](#)
- braid_SetRelTol
 - General Interface routines, [35](#)
- braid_SetSpatialCoarsen
 - General Interface routines, [35](#)
- braid_SetSpatialRefine
 - General Interface routines, [36](#)
- braid_SetTemporalNorm
 - General Interface routines, [36](#)
- braid_SetTightxTol
 - General Interface routines, [36](#)
- braid_SplitCommworld
 - General Interface routines, [36](#)
- braid_TestAll
 - XBraid test routines, [46](#)
- braid_TestBuf
 - XBraid test routines, [47](#)
- braid_TestClone
 - XBraid test routines, [47](#)
- braid_TestCoarsenRefine
 - XBraid test routines, [48](#)
- braid_TestInitAccess
 - XBraid test routines, [48](#)
- braid_TestSpatialNorm
 - XBraid test routines, [49](#)
- braid_TestSum
 - XBraid test routines, [49](#)
- braid_Vector
 - User-written routines, [28](#)
- [braid_defs.h](#), [68](#)
 - [_braid_CTAlloc](#), [68](#)
 - [_braid_TAlloc](#), [68](#)
 - [_braid_TFree](#), [68](#)
 - [_braid_TReAlloc](#), [68](#)
 - [_braid_error_flag](#), [69](#)
 - [braid_Int](#), [68](#)
 - [braid_Real](#), [68](#)
- [braid_status.h](#), [69](#)
- [braid_test.h](#), [70](#)
- buffer
 - [_braid_CommHandle](#), [53](#)
- bufpack
 - [_braid_Core](#), [55](#)
- bufsize
 - [_braid_Core](#), [56](#)
- bufunpack
 - [_braid_Core](#), [56](#)
- c_tprior
 - [_braid_CoarsenRefStatus](#), [53](#)
- c_tstop
 - [_braid_CoarsenRefStatus](#), [53](#)
- cfactor
 - [_braid_Grid](#), [59](#)
- cfactors
 - [_braid_Core](#), [56](#)
- cfdefault
 - [_braid_Core](#), [56](#)
- clone
 - [_braid_Core](#), [56](#)
- clower

- [_braid_Grid, 59](#)
- coarsen
 - [_braid_Core, 56](#)
- comm
 - [_braid_Core, 56](#)
- comm_world
 - [_braid_Core, 56](#)
- cstatus
 - [_braid_Core, 56](#)
- cupper
 - [_braid_Grid, 59](#)
- done
 - [_braid_AccessStatus, 51](#)
- f_tprior
 - [_braid_CoarsenRefStatus, 53](#)
- f_tstop
 - [_braid_CoarsenRefStatus, 53](#)
- fmg
 - [_braid_Core, 56](#)
- free
 - [_braid_Core, 56](#)
- General Interface routines, [30](#)
 - [braid_Core, 30](#)
 - [braid_Destroy, 31](#)
 - [braid_Drive, 32](#)
 - [braid_GetNumIter, 32](#)
 - [braid_GetRNorm, 32](#)
 - [braid_Init, 32](#)
 - [braid_PrintStats, 33](#)
 - [braid_SetAbsTol, 33](#)
 - [braid_SetAccessLevel, 33](#)
 - [braid_SetCFactor, 33](#)
 - [braid_SetFMG, 34](#)
 - [braid_SetLoosexTol, 34](#)
 - [braid_SetMaxIter, 34](#)
 - [braid_SetMaxLevels, 34](#)
 - [braid_SetMinCoarse, 34](#)
 - [braid_SetNFMGVcyc, 34](#)
 - [braid_SetNRelax, 35](#)
 - [braid_SetPrintFile, 35](#)
 - [braid_SetPrintLevel, 35](#)
 - [braid_SetRelTol, 35](#)
 - [braid_SetSpatialCoarsen, 35](#)
 - [braid_SetSpatialRefine, 36](#)
 - [braid_SetTemporalNorm, 36](#)
 - [braid_SetTightxTol, 36](#)
 - [braid_SplitCommworld, 36](#)
- globaltime
 - [_braid_Core, 56](#)
- grids
 - [_braid_Core, 56](#)
- gupper
 - [_braid_Core, 56](#)
 - [_braid_Grid, 59](#)
- ilower
 - [_braid_Grid, 59](#)
- init
 - [_braid_Core, 56](#)
- iter
 - [_braid_AccessStatus, 51](#)
- iupper
 - [_braid_Grid, 59](#)
- level
 - [_braid_AccessStatus, 51](#)
 - [_braid_Grid, 59](#)
- localtime
 - [_braid_Core, 57](#)
- loose
 - [_braid_AccuracyHandle, 52](#)
- matchF
 - [_braid_AccuracyHandle, 52](#)
- max_iter
 - [_braid_Core, 57](#)
- max_levels
 - [_braid_Core, 57](#)
- min_coarse
 - [_braid_Core, 57](#)
- ncpoints
 - [_braid_Grid, 59](#)
- nfmvg_Vcyc
 - [_braid_Core, 57](#)
- niter
 - [_braid_Core, 57](#)
- nlevels
 - [_braid_Core, 57](#)
- nrdefault
 - [_braid_Core, 57](#)
- nrels
 - [_braid_Core, 57](#)
- ntime
 - [_braid_Core, 57](#)
- num_requests
 - [_braid_CommHandle, 54](#)
- old_value
 - [_braid_AccuracyHandle, 52](#)
- phi
 - [_braid_Core, 57](#)
- print_level
 - [_braid_Core, 57](#)
- pstatus
 - [_braid_Core, 57](#)

- recv_handle
 - _braid_Grid, 60
- recv_index
 - _braid_Grid, 60
- refine
 - _braid_Core, 57
- request_type
 - _braid_CommHandle, 54
- requests
 - _braid_CommHandle, 54
- rfactor
 - _braid_PhiStatus, 61
- rfactors
 - _braid_Core, 58
- rnorm
 - _braid_AccessStatus, 51
 - _braid_Core, 58
- rtol
 - _braid_Core, 58
- send_handle
 - _braid_Grid, 60
- send_index
 - _braid_Grid, 60
- spatialnorm
 - _braid_Core, 58
- status
 - _braid_CommHandle, 54
- sum
 - _braid_Core, 58
- t
 - _braid_AccessStatus, 51
- ta
 - _braid_Grid, 60
- ta_alloc
 - _braid_Grid, 60
- tight
 - _braid_AccuracyHandle, 52
- tight_used
 - _braid_AccuracyHandle, 52
- tnorm
 - _braid_Core, 58
- tnorm_a
 - _braid_Core, 58
- tol
 - _braid_Core, 58
- tstart
 - _braid_CoarsenRefStatus, 53
 - _braid_Core, 58
 - _braid_PhiStatus, 61
- tstop
 - _braid_Core, 58
 - _braid_PhiStatus, 61
- ua
 - _braid_Grid, 60
- ua_alloc
 - _braid_Grid, 60
- User interface routines, 29
- User-written routines, 25
 - braid_App, 26
 - braid_PtFcnAccess, 26
 - braid_PtFcnBufPack, 26
 - braid_PtFcnBufSize, 26
 - braid_PtFcnBufUnpack, 27
 - braid_PtFcnClone, 27
 - braid_PtFcnCoarsen, 27
 - braid_PtFcnFree, 27
 - braid_PtFcnInit, 27
 - braid_PtFcnPhi, 27
 - braid_PtFcnRefine, 27
 - braid_PtFcnSpatialNorm, 27
 - braid_PtFcnSum, 28
 - braid_Vector, 28
- util.h, 70
 - _braid_Max, 71
 - _braid_ParFprintfFlush, 71
 - _braid_ProjectInterval, 71
 - _braid_SetAccuracy, 71
 - _braid_printf, 71
- va
 - _braid_Grid, 60
- va_alloc
 - _braid_Grid, 60
- value
 - _braid_AccuracyHandle, 52
- vector_ptr
 - _braid_CommHandle, 54
- wa
 - _braid_Grid, 60
- wa_alloc
 - _braid_Grid, 60
- wrapper_test
 - _braid_AccessStatus, 51
- XBraid status routines, 38
 - _braid_AccessStatusDestroy, 39
 - _braid_AccessStatusInit, 39
 - _braid_CoarsenRefStatusDestroy, 39
 - _braid_CoarsenRefStatusInit, 39
 - _braid_PhiStatusDestroy, 41
 - _braid_PhiStatusInit, 41
 - _braid_StatusElt, 39
 - braid_AccessStatus, 39
 - braid_AccessStatusGetDone, 41
 - braid_AccessStatusGetIter, 41
 - braid_AccessStatusGetLevel, 41

- braid_AccessStatusGetResidual, [42](#)
- braid_AccessStatusGetT, [42](#)
- braid_AccessStatusGetTILD, [42](#)
- braid_AccessStatusGetWrapperTest, [42](#)
- braid_CoarsenRefStatus, [39](#)
- braid_CoarsenRefStatusGetCTprior, [42](#)
- braid_CoarsenRefStatusGetCTstop, [43](#)
- braid_CoarsenRefStatusGetFTPrior, [43](#)
- braid_CoarsenRefStatusGetFTstop, [43](#)
- braid_CoarsenRefStatusGetTpriorTstop, [43](#)
- braid_CoarsenRefStatusGetTstart, [43](#)
- braid_PhiStatus, [39](#)
- braid_PhiStatusGetAccuracy, [44](#)
- braid_PhiStatusGetTstart, [44](#)
- braid_PhiStatusGetTstartTstop, [44](#)
- braid_PhiStatusGetTstop, [44](#)
- braid_PhiStatusSetRFactor, [44](#)
- XBraid test routines, [46](#)
 - braid_TestAll, [46](#)
 - braid_TestBuf, [47](#)
 - braid_TestClone, [47](#)
 - braid_TestCoarsenRefine, [48](#)
 - braid_TestInitAccess, [48](#)
 - braid_TestSpatialNorm, [49](#)
 - braid_TestSum, [49](#)